Symbolic Analysis of C Binaries

A Thesis
Presented to
The Established Interdisciplinary Committee for Mathematics and Computer Science
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Thomas Ulmer

May 2024

Approved for the Committee
(Mathematics and Computer Science)

Dylan McNamee	David Meyer

Acknowledgments

- Thank you to Langston Barrett at Galois for very patiently answering my many questions about Crucible.
- Thank you to my advisors Dylan and David for their edits and suggestions, this thesis would be incomprehensible without them.
- Thank you to my friends for listening to my ramblings about Haskell even if they don't understand it.
- Thank you to my family for their advice and support on the thesis process, even if it all sounds like Greek to them.
- Thank you to Ozymandias Juarez for reminding me that standards come from within. Who would do this if not I?

Table of Contents

Abstract

Within the field of static analysis, symbolic analysis is a powerful tool for verfying the behavior of computer programs. The work presented here extends an existing library for symbolic analysis to operate on binary programs in addition to the structure-rich source code or intermediate representation it already supports. This addition brings symbolic analysis into scope as a possible tool for reverse engineering binaries, as well as confirming expected behavior between a model and a binary or source code and a binary. The code of this thesis is novel in that it brings P-Code into direct communication with this symbolic ecosystem, allowing for testing of the ecosystem itself via differential testing as well the more typical uses such as compiler verification. In addition, it brings new symbolic analysis to new architectures in a uniform way through P-Code rather than being supported individually as is the case for those currently supported.

Introduction

> "I don't see much sense in that," said Rabbit.
> "No," said Pooh humbly, "there isn't. But there was going to be when I began it. It's just that something happened to it on the way."
>
> A. A. Milne

As what we demand of computers has grown, the size and complexity of programs has also dramatically increased. In general a modern programmer's code is judged on correctness, efficiency, and readability. All three metrics are important, but it is generally agreed that they are to be prioritized in the above order. However even with the correctness of program top of mind, as programs grow past what a single person can write or what a single person can have in their head at one time, it grows harder to ensure correctness in all cases. Readability can be generally improved by following a consistent style, careful naming, and thorough commenting. Efficiency is difficult, but there are well known techniques and, importantly, there are metrics with which one can experiment. Correctness is harder to ensure in general, as it is not a property of the code per se, as is the case with efficiency. It can't be easily reduced to a metric. This is because a program's correctness is a measure of distance between what the code does, and what it is intended to do. The former is a mechanical expansion of the code according to the rules that govern the coding language and machine. The latter however, is not so easily expressed formally.

A program that has unintended behavior is still a functional program. It simply calculates something other than what the programmer expects. Often that alternate calculation can seem nonsensical when viewed through the lens of the expected calculation. This poses a problem. Naturally we want our code to do what we intend it to. The question then is how can we guard ourselves, as programmers, from unintended behavior.

Traditionally, the solution has been testing. If one knows what the program is designed to do, then one can run the program on specified inputs, and ensure that the outputs and behavior match what is expected. The simplicity of this method, and the length of time that it has been the dominant method, have lead to very streamlined methods of testing. For instance, many programming languages allow the programmer to include a suite of tests in the source code so that when they compile, a potentially very large set of tests is run automatically and the programmer is informed of their output with the emphasis placed on if and which tests failed. However even for

trivially simple programs, these tests cannot be exhaustive. A program that takes a 32 bit number and adds one before returning the sum, for instance, requires 4.2 billion cases. The speed of modern computers could accomplish that in a second. However the scaling in not in our favor. For each such input, one must consider all combinations of possible inputs, which will scale exponentially. Clearly, exhaustive concrete testing is a fool's errand.

This comparison is not completely fair. Most programs have internal structure designed by the programmer. The most fundamental kind of structure is repetition. A program that takes two numbers, squares them both, and returns both likely uses the same mechanism twice. So then the programmer or the tester could make the judgment that so long as the first number is correct in all cases, we don't need to test the second. This judgment, informed by looking at the source code may very well be reasonable. However we soon run into the same difficulties. When are these judgments safe to make? It turns out that knowing that a piece of code is safe not to test is really the same problem as knowing that a piece of code is correct in the first place. This higher level version of the issue at hand is often easier, but has the same challenges of scale when programs get complex.

Complexity-reducing techniques exist: one can test parts of the program independently or take other simplifying actions, but those techniques are not powerful enough to work in all cases. The best one can do is consider only programs written within some self-imposed constraints. However productive work is not done solely within these bounds, and one needs to work outside of them at times. It would be good if we could feel safe while doing so.

So far the approach we have discussed is tantamount to bringing the conceptual model of the program down to the written version (the source code). We specialize the model to a specific case, or small band of cases, and we check that the two agree. However, what about the other direction? Can we lift the written concrete version to a higher level of abstraction, and do our comparison there? The answer, unsurprisingly, is sometimes.

Static analysis is the general term for the collection of problems comparing written code and mental models in the direction from code to model. The lifting problem just described is called **model synthesis**. We can imaging defining a language of very precise meanings, and writing our conceptual idea in that language. A very natural language exists: mathematics. The central conceit of mathematics is that the base on which it is built is simple enough to be checked for soundness, and that we agree on it. Then new math can be build atop the old, and it is as trusted as that which it is build on and of. Philosophy of math aside, if we take mathematical notation as a concrete language we trust, and we write down what we want in math, then we are halfway there. We have two written versions of what is supposedly the same idea, the same computation. The problem is that they are in different languages. To translate a program that can be run to a set of mathematical statements that encode its effects is the purpose of static analysis. Better yet, it turns out we can often do that mechanically.

0.1 Layout of this Document

Here we give an overview of this thesis, as the division and order of chapters requires some explanation.

The first chapter contains explanations of several conceptions of computation that will underpin the remainder of the thesis. They are presented in the first chapter in some specificity as they will be assumed in all the chapters after.

After laying out the underlying ideas, next is a chapter that details P-Code, the representation of programs that we take as input. This description is more surface level, as the exact details of each piece are not critical. Instead the goal is to introduce P-Code and place it in the context of representations of computer programs, particularly in order to highlight the ways in which it differs from the models presented in the Prerequisites chapter, as those differences must be addressed by the code artifact of this thesis.

Following that, a chapter about symbolic analysis at a high level. This chapter explores some of the goals and difficulties that result from the motivations mentioned above. The implementation details of the problems are not explored, but a simplified version of symbolic analysis is presented in parts over the course of the chapter to motivate the ideas and reveal the obstacles that result.

After the goal and the high level idea is presented in the symbolic analysis chapter, the next chapter goes into more depth about the actual code artifact of this thesis. A number of the details of the symbolic analysis are not handled explicitly and instead are left to the library that provides many of the components and tools that we use. However a number of problems appear from the differences between P-Code and the other models presented. This chapter details some of them and their respective solutions.

After the chapter about the code artifact, the reader has a complete idea of what this thesis set out to do, and to some extent how it was achieved. However the technical work of the thesis was concentrated not on what the code does, designing algorithms and the like, but instead on the details of how to make each small step function. There is a tremendous amount of code complexity introduced by the library that provides much of the symbolic analysis toolkit we want to use. In order to use this library, a great deal of esoteric theory and paradigms needed to be employed. The following chapter, giving an account of the basics of type theory in the context of the lambda calculus, lays the groundwork for the most interesting aspects of programming in these paradigms.

The final chapter presents some of the technical considerations of programming in this context. In particular, some features of the language that require a change of mindset and the specialized tools required by the symbolic analysis library. These specialized tools are presented in relation to the account presented in the type theory chapter, as much of the difficulty in implementation that stems from the ways in which the language differs from the standard formulation presented in the prior chapter.

The following diagram illustrates how our work fits in existing technologies and the three models of computation presented in the next chapter.

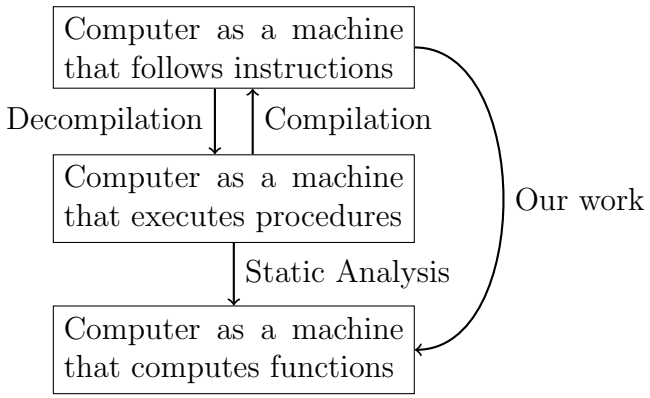


Figure 1: Overview of Related Technologies

Chapter 1

Prerequisites

> What's reality? I don't know. When my bird was looking at my computer monitor I thought "That bird has no idea what he's looking at." And yet what does the bird do? Does he panic? No, he can't really panic, he just does the best he can. Is he able to live in a world where he's so ignorant? Well, he doesn't really have a choice. The bird is okay even though he doesn't understand the world. You're that bird looking at the monitor, and you're thinking to yourself, I can figure this out. Maybe you have some bird ideas. Maybe that's the best you can do.
>
> Terry Davis

This chapter contains the background information we deem necessary to understand the mechanics and the motivation to this thesis. Particularly it lays out the model of computation that underpins the reasoning in this thesis for the lay reader. After establishing a general understanding of what a computer is, at the level we care about, this chapter contains three sections that detail the three most relevant ways that programmers think about computers and go about programming them: as a machine that follows instructions, as a machine that follows procedures, and as a machine that calculates functions. Again this treatment is focused on the conceptual issues and will avoid getting bogged down in the exact specifics of how these views differ and are implemented in reality.

1.1 Computation Model

What is a computer, and how does one instruct it to do something? Those are the questions this chapter sets out to answer. This section exists to answer the first one. Here we give a definition of a **model of computation**. A model of computation defines the parts of a computer, how they interact, and what basic actions happen within and between them. A complete model has a comprehensive set of rules such

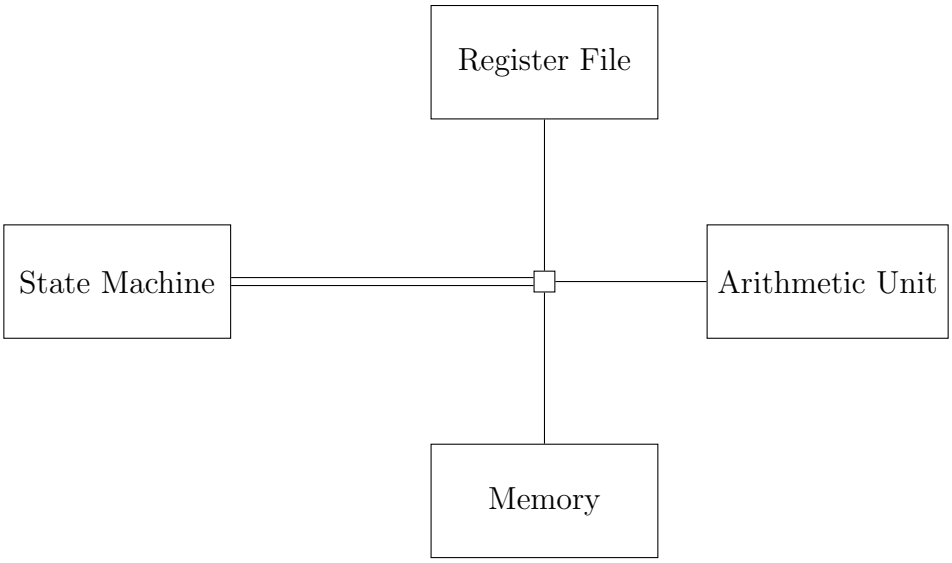


Figure 1.1: Computational Model

that there is no state or action that is not modeled. However we will see that there is some use in under-specifying a computer, and leaving space for undefined behavior. Not only does it allow for a simpler presentation that will be more intuitive to a lay reader, but this under-specification is extremely common in the real world.

The model of computation presented here is a combination of a Von Neumann machine, the standard informal RAM model, and a dash of the shared model of the C family of languages.

Our computer model consists of four parts. They are the arithmetic unit, the memory unit, the register unit, and the state machine. One can assume that these components are all interconnected, and not worry about the exact details of how they communicate. Each of these components will be presented in turn, and their combination will be explained afterwards. Before the description can begin however, we must explain undefined behavior.

1.1.1 Undefined Behavior

When we say that some state or action of the computer resulting in **undefined behavior**, we mean just that. The model does not specify what occurs. This can be slightly unintuitive, but an action resulting in undefined behavior could do anything. Canonical examples could be performing some sensible default action (producing zero for a calculated value perhaps) bringing the execution of the computer to a halt (optionally with some sort of external sign that something has gone wrong), or even causing the physical device to catch fire or explode. The key insight is that anything *could* happen, and the same thing doesn't have to happen every time, even if the rest of the system is identical. Naturally undefined behavior is a bad mix with expecting the computer to reliably perform meaningful work. As such, most programs strive to

avoid it, however it remains a core part of many models of computation, including the one presented here.

1.1.2 Register Unit

The register unit, also called the register file, is responsible for storing data for short periods of time. The register file contains some fixed, small number of entries, called **registers**, which can hold some small fixed quantity of data. Each register has the same number of bits, and thus the same capacity for storing information. The registers are numbered from zero to the appropriate number. The exact number is irrelevant here, but there are a fixed number for each machine.

The register unit overall supports two operations. The first is a **read**. A read returns the binary data contained in a register. In combination with other units, reads are used to make data available (as a copy) to other units. So to summarize, a read with an indicated register leaves the contents of all the registers the same, but makes the contents of the indicated register available to the rest of the system.

The second operation is the reverse. A **write** updates the contents of an indicated register with some data supplied by the rest of the system. And following read on an effected register of a write will produce the new data rather than the contents before the write. Again the other registers are not affected.

Reading a register before writing it produces undefined behavior. The register must be initialized with some known value via a write before it is safe to access.

With this the first sub-system is realized. The register unit lets the computer save and retrieve data. The next question is what can the computer do with that data?

1.1.3 Arithmetic Unit

The arithmetic unit is responsible for performing computations as humans would recognize it. It contains some fixed set of operations. Provided with an operation and the correct number of pieces of input data (generally two), it produces a new piece of data according to the operation. Common operations include elementary operations on integers (addition, subtraction, multiplication, and division), along with binary operations such as `and`, `or`, exclusive or (`xor`), `not`, and shifts to the left and right (`shl`, `shr`). Some combinations of inputs and operations may produce undefined behavior. A canonical example could be attempting to divide by zero.

1.1.4 Memory Unit

The memory unit is very similar to the register unit. However its internal arrangement, strengths and limitations are different. The memory unit supports two operations analogous to the register unit's read and write. They are **load** and **store**.

A memory load is a read from memory. It takes an **address** and (often implicitly) a length, and copies the binary data of that length starting at that address, making it available to the rest of the system. A memory store is the reverse, taking a piece of data of some length and an address and saving the data at the given address for

future accesses. The same relationship between sequential reads and writes exists between loads and stores. Again loading before storing at the same address produces undefined behavior.

There is an initial point of nuance that separates memory from registers. In the register file, registers are disjoint. One can either access some register or the next, but not part of a register or an interval that takes parts of two registers. Memory on the other hand is a contiguous sequence of addresses. So if a load at address a for some length l might produce some string of bits $b_0 b_1 \ldots b_{l-1}$, the load with the same address but half the length would produce $b_0 b_1 \ldots b_{l/2-1}$. A load with address $a + (l/2)$ and length $l/2$ would produce $b_{l/2} b_{l/2+1} \ldots b_{l-1}$. There are a number of small details that need to be worked out about the full generality of memory access and updates, but we will elide that here for brevity. This is just to give a sense of one way in which memory is more flexible than registers. The other way in which they differ leads us to the final unit. A read or write to a register requires the register in question to be indicated in the operation. For any individual register unit operation, that indicator value is fixed, and doesn't change during the course of the execution of a program. A memory load or store does not have this limitation. For instance, the address of a load could be obtained from a register. This means that as the register changes value over the course of the execution of a program, the location that the data is copied from will also change. This is not possible with registers, one cannot select the index of the effected register with another register, or any other changing value in the system.

The register file has some fixed small number of registers (often 64 or so). When an instruction involves a register, it is indicated by it's number. The total number of registers is thus bounded by the number of bits in the instruction encoding. The **addressable range** of computer is full set of memory addresses accessible. This is often the number of addresses one can represent with a single register. Note that many instruction encoding use 5 or 6 bits to indicate registers, but a register might be 32 or 64 bits wide. Thus the number of addresses one can use in memory is many orders of magnitude bigger than the number of registers.

The memory unit has an extremely large number of data locations, often thought of as either finite but exceedingly large or infinite, depending on the analysis. These data locations are arranged sequentially. The memory is a sequence of cells, each associated with an address. We say that the natural numbers index memory, i.e. each natural number is associated with a memory cell, and the cells are laid out in a manner that captures the layout of the naturals. In short this means that adjacent naturals should have adjacent cells. In most machines, cells can contain a single byte (8 bits, or a number equal to or between -127 and 128).

Consider a store at address a of length l of some string of bits $b_0 b_1 \ldots b_{l-1}$. A following load at the same address and with the same length will produce the exact same bit string as expected.

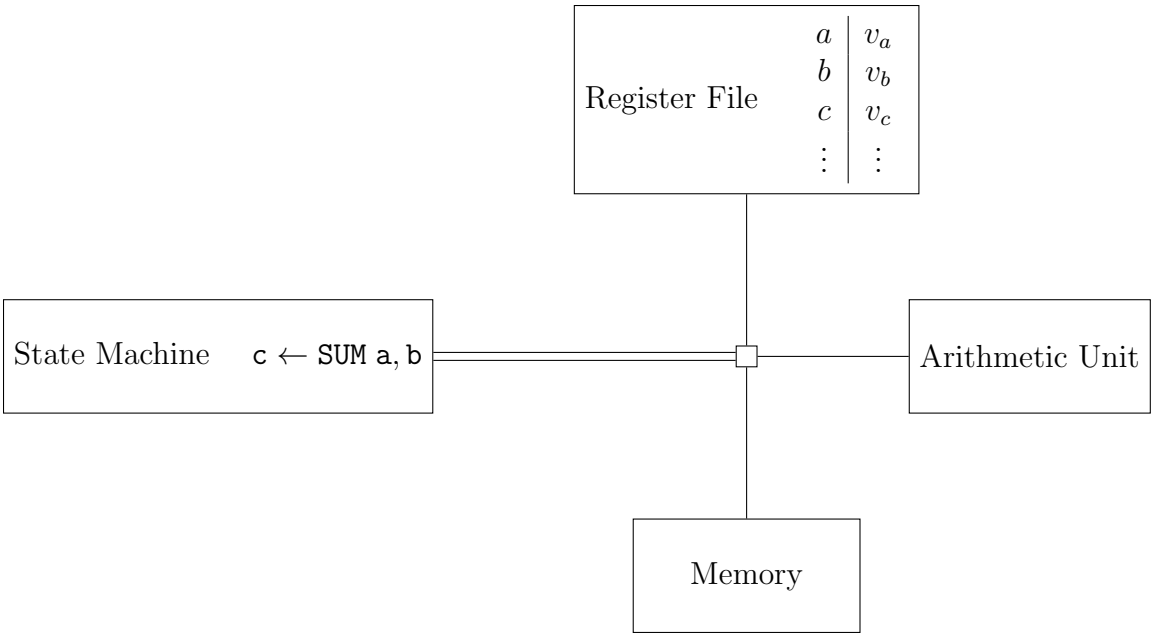
Now consider two stores $(a, l, \vec{x})$ and $(a + l, l, \vec{y})$. A load with length l and address $a + (l/2)$ produces the second half of $\vec{x}$ and the first half of $\vec{y}$.

As with registers, a load that accesses cells that have not yet been stored to produce undefined behavior. It is often convenient to think of cells and registers

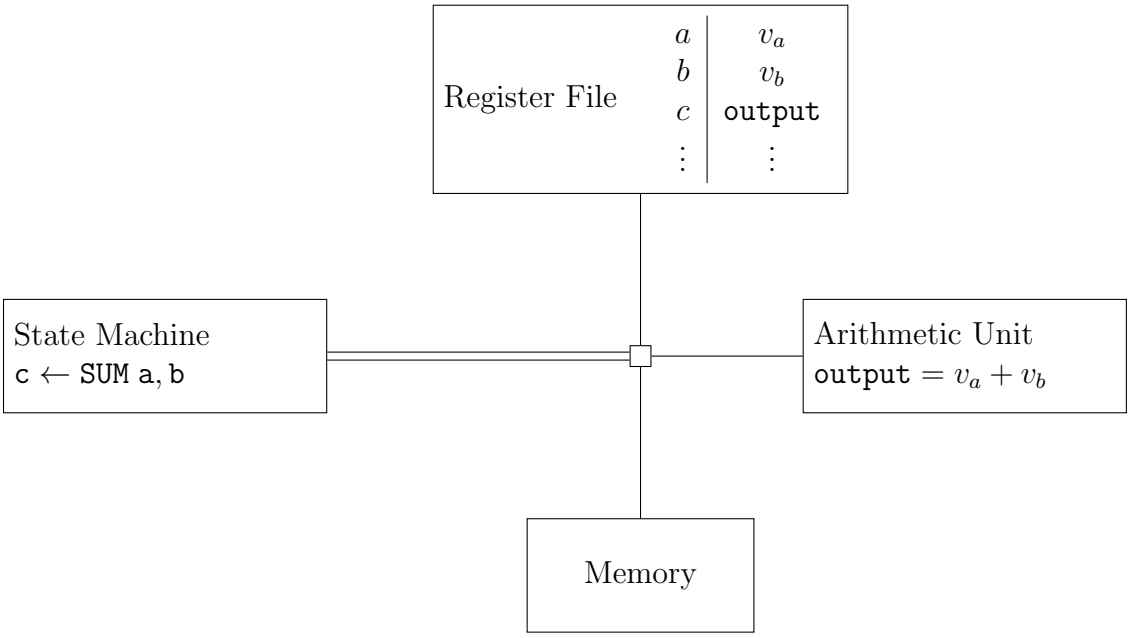
of being of the same width. This is not true of modern computers, but it simplifies analysis. The data effect diagrams at the end of this section assume that registers and memory cells are the same size (one byte). This ensures we do not need to perform any arithmetic on addresses and that our loads and stores only need to access a single cell. In reality registers are generally four to eight times wider than memory cells. For instance, a byte-addressable memory with 8-byte registers is common for modern consumer devices. In this case if one wants to store the contents of two registers in memory side by side, the first will occupy cells a through $a + 7$ and the second $a + 8$ through $a + 15$. Note that storing into memory simply over-writes the previous contents, it does not insert new cells or otherwise shift cells.

1.1.5 State Machine

The final unit can be considered the brain of the system. The state machine does several things. The most important of which is direct the other units. The state machine is exactly what it is called, a state and rules about how that state changes according to a sequence of inputs. The sequence of inputs is divided into **instructions**. An instruction is a small unit of computation. It is composed of some operation (arithmetic, memory, register, generally a combination), and the details of how to connect the units together. For example, an instruction that says to take the two registers a and b and place their sum in register c. The state machine interprets this instruction, and causes the register unit to perform a read of a and a read of b, routes those new pieces of data to the inputs of the arithmetic unit, and indicates that the operation to perform on them is addition. It then routes the output data from the arithmetic unit back to the register unit, and produces a write operation on the register c to save the value.



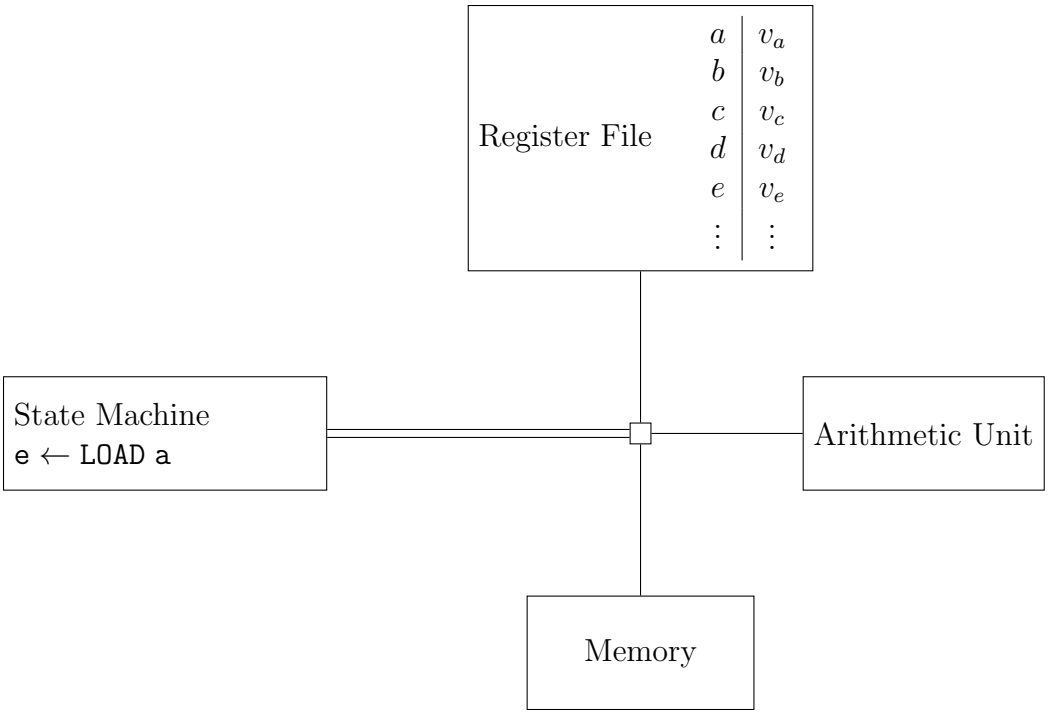
(a) Instruction Interpretation



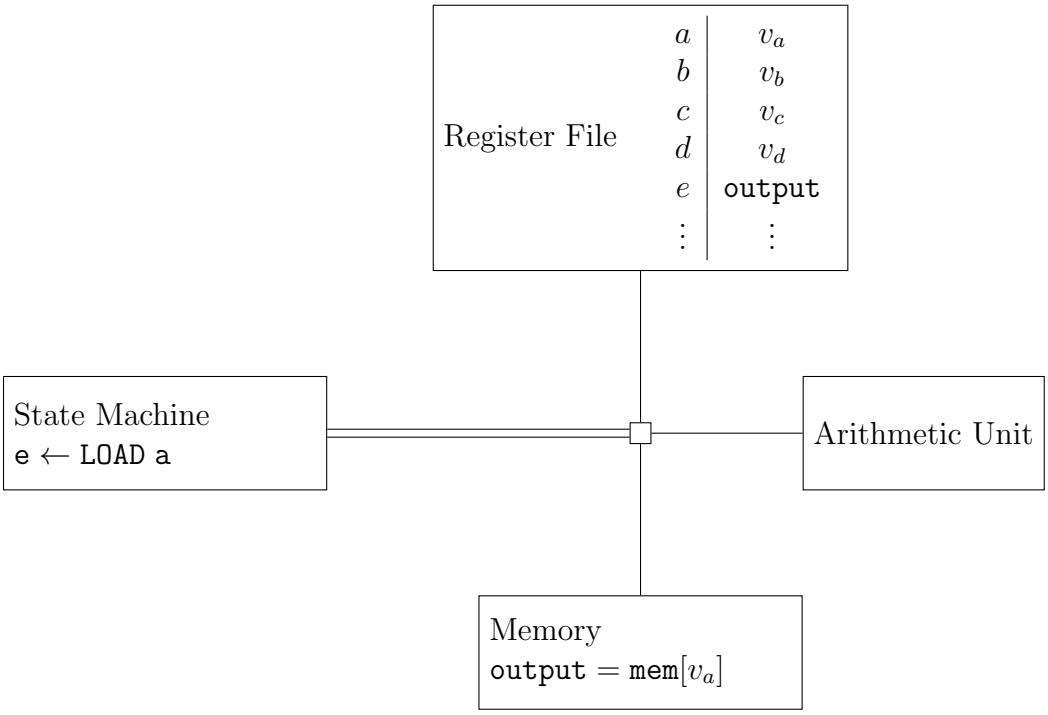
(b) Instruction Execution

The other common kind of instruction is a memory instruction. Our model is limited to two kinds. The first performs a load from memory, placing the resulting value in a register. The second does the reverse, reading from one register for a value, and another for the address, and storing the value at the address in memory. These memory instructions are required, as our model only allows arithmetic operations to take in data from and output to registers. Other models exist that are more permissive.

Consider a load into register e with the address of register a.



(a) Instruction Interpretation



(b) Instruction Execution

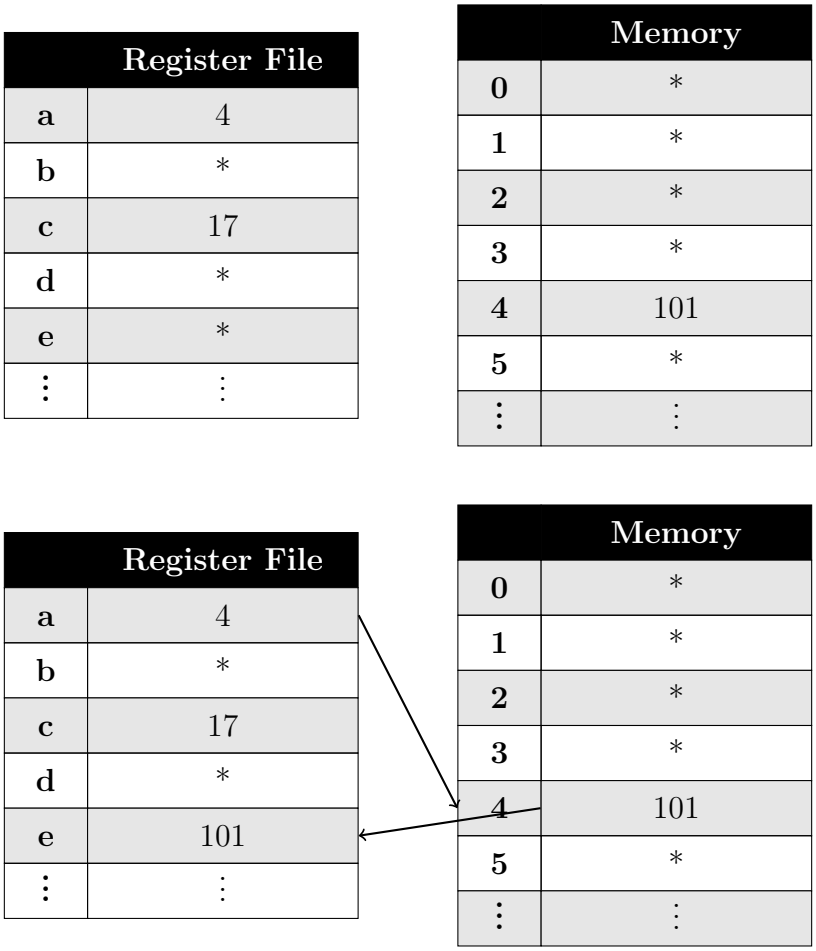


Figure 1.4: Data Effect of `LOAD e, a`

This load from memory demonstrates that data can be moved from memory to registers. Note that the register used as the destination of the data and the register that contains the address from which to pull data from are both selected by the programmer. In the opposite direction there are stores to memory.

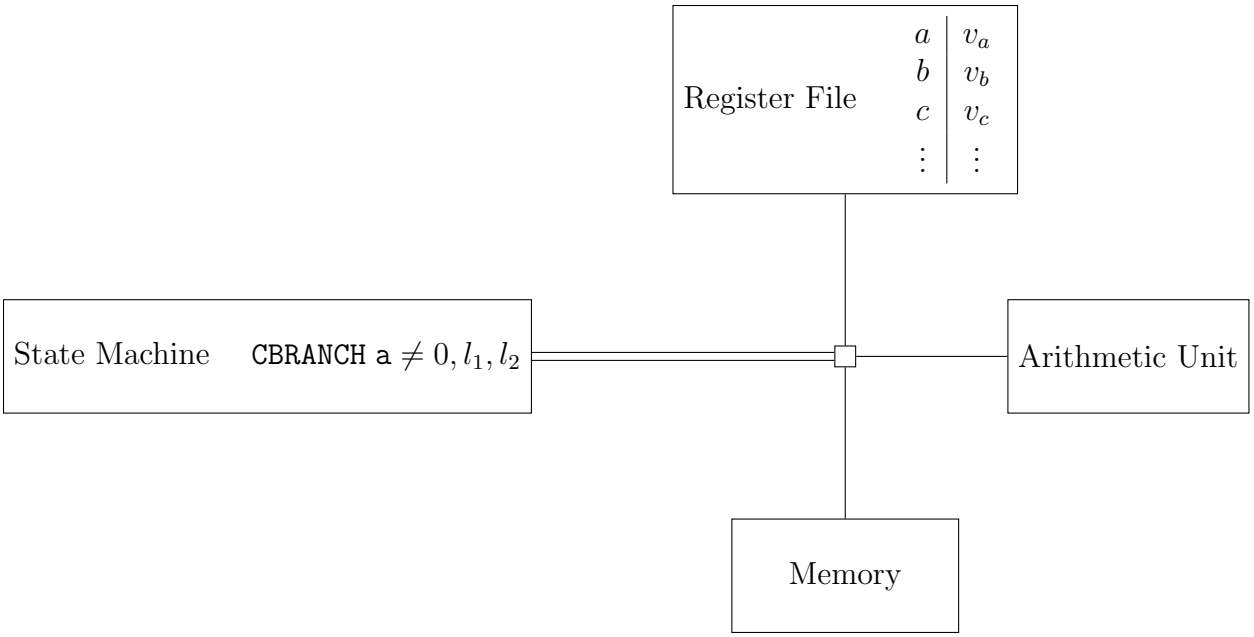
The next natural question is "where do instructions come from?" The answer is that instructions are encoded as a sequence of bits, and are stored in memory. So the state machine first performs a load from memory at some address stored in an indicated register called the **program counter**, and then interprets the result as an instruction and performs further operations from there. After the instruction has been completed, the program counter is incremented to point to the next instruction in memory sequentially, and the process is repeated. Thus if there is a sequence of instructions stored in memory, the state machine will pass over them, executing them one at a time in sequence.

The **program** refers to the sequence of instructions placed in memory that define the operation of the computer. The program is considered to be the initial contents of the memory unit, and execution of the program starts at the first instruction.

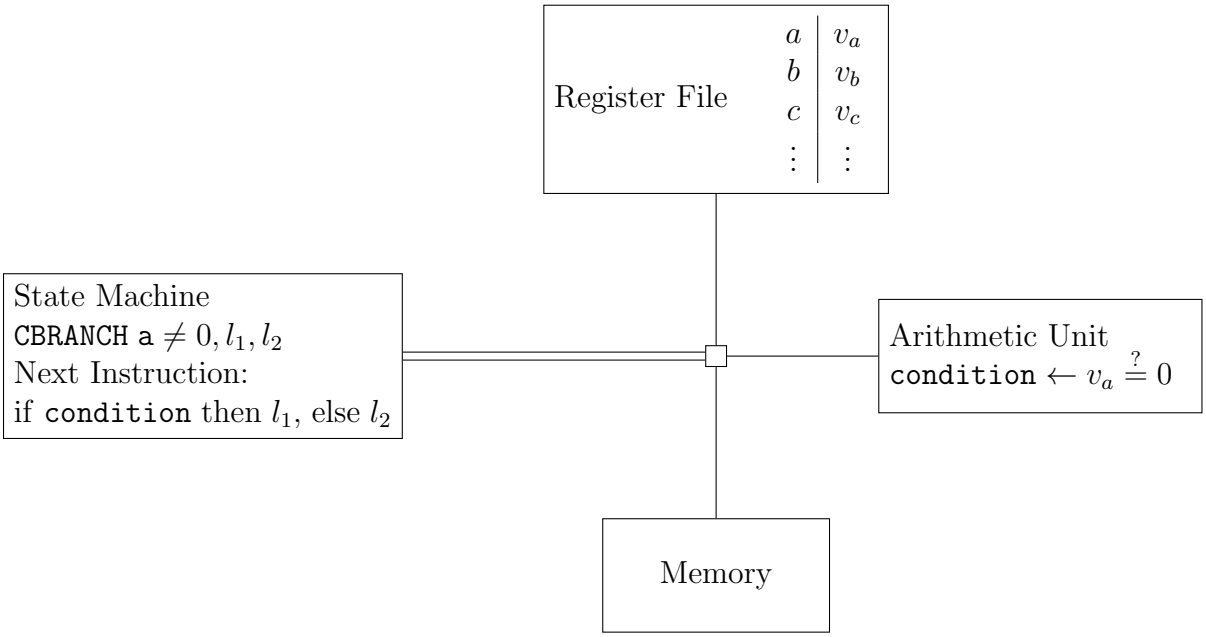
Note that for a fixed computer, the choice and layout of instructions in memory fully describes the computation performed by the computer. At its core, programming is the creation of a program, particularly one that describes some computation of interest to humans.

Finally there are a few special instructions that manipulate this program counter. They are collectively referred to as **control flow instructions**, as they indicate how the execution passes through the program. In normal operation the state machine executes every instruction in order. However it can be useful to have more flexibility. The most basic kind of control flow is a **branch** (sometimes called a jump). A branch contains an address, either directly or as an offset from the current program counter. When the state machine executes a branch, the next instruction that is executed is the one indicated by the branch, which may not be the instruction immediately following the branch instruction that was just executed. In addition, there are conditional branches, which check some true or false property (often if a register is zero), and either execute the branch or continue to the following instruction depending on the result of the condition. Finally there are indirect branches, which take an address in a register and cause execution to begin at the memory located there. This can be thought of as writing the provided address to the program counter register.

Consider a condition branch, which directs program flow to the address l_1 if register a is zero, and l_2 otherwise.



(a) Instruction Interpretation



(b) Instruction Execution

1.1.6 Examples

This section gives some examples of the data effects of a selection of instructions in our computation model. These examples elide details of how operations function and presents the way they change the state of the computer, namely be altering registers or memory.

	Register File
a	42
b	*
c	17
d	*
⋮	⋮

	Memory
0	*
1	*
⋮	⋮

	Register File
a	42
b	59
c	17
d	*
⋮	⋮

	Memory
0	*
1	*
⋮	⋮

Figure 1.6: Data Effect of `ADD c, a, b`

This register-only addition instruction is representative of all arithmetic and logical operations. Said instructions perform some binary operation on the contents of two registers and writes the contents to a register. The involved registers are selected by the programmer. In the opposite direction of the memory load presented above, there are memory stores.

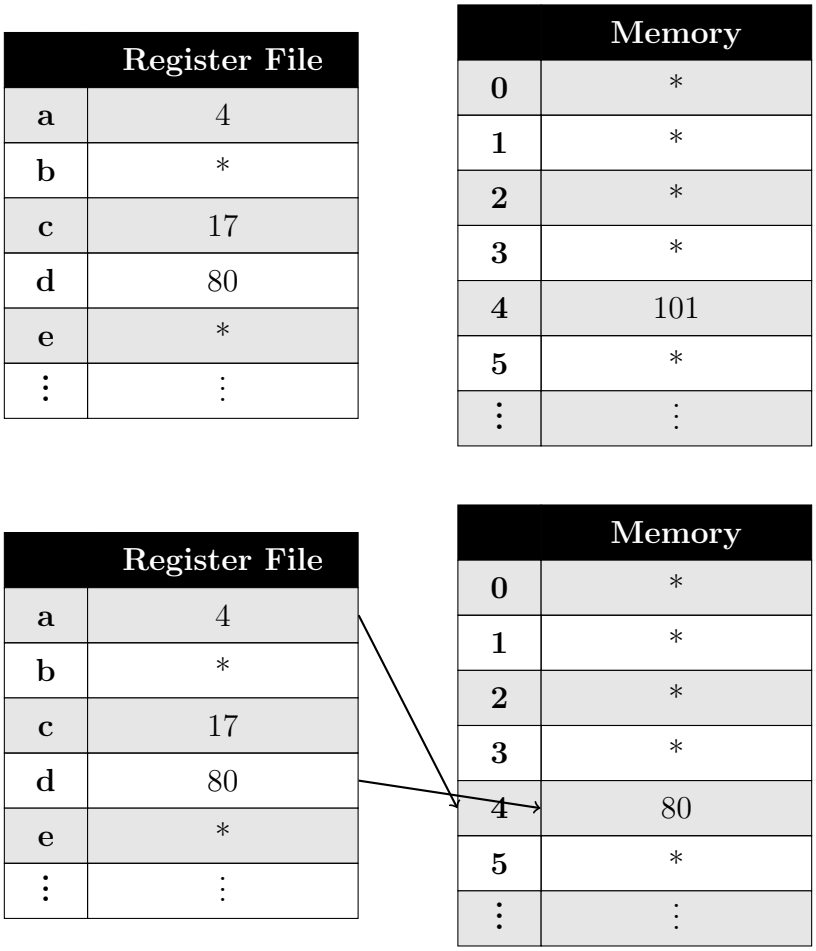


Figure 1.7: Data Effect of `STORE d, a`

1.2 Assembly

Now we have a complete (if quite unrealistic) description of a computer, and we know that to make the computer perform varied calculations, it suffices to change the program that starts in memory. From this point on, the computer is thought to be fixed, and the problem becomes how to produce a program (i.e. a sequence of instructions) that describes the computation we desire as programmers.

Assembly language is generally considered the lowest form of human readable code, and the most basic programming language. It is simply a nicer representation of the sequence of bits for each instruction. Instead of forcing the programmer to input each instruction is some impenetrable sequence of digits, the meaning of which is unclear, assembly gives mnemonic names to each instruction, along with the other elements that the programmer needs to describe a computation. This thesis is primarily considered with the assembly language associated with the RISC-V specification. The exact details are not critical, but a full reference of the relevant parts of the

RISC-V specification are found in the appendix A.

Take for instance our example of adding two registers and storing the result in a third. In RISC-V's assembly, we could write `ADD x1, x2, x3`, which computes the sum of registers `x2` and `x3` and stores the result in `x1`. In addition, when it comes to control flow, assembly allows the program to give names to locations in the program called **labels**. So one can write `j loop1` where `loop1` is a label defined elsewhere in the code. `j` is short for jump, which is what RISC-V calls unconditional branches.

Assembly is notable because is completely transparent. If one has assembly, they can immediately produce the actual sequence of bits that a computer expects, and vice versa. The human readable assembly is recoverable from the binary program that the computer actually runs. Assembly thus represents a completely unambiguous representation of the program. This is notable, as it is not true of most programming languages. Almost all other languages exist to abstract away the tedious details of the computer and allow the programmer to write code at a higher level of abstraction. This includes automatically saving and restoring data in different contexts, named data locations, more complex control flow, and data representing non-trivial encoding of bits. It is important to note that these are all representable in assembly, in fact most languages are translated (**compiled**) into assembly or directly into machine code (the numeric encoding the computer expects). However they are tedious, error prone, and complex to implement well.

1.3 Imperative Languages

The next kind of description of a computation is **imperative** languages. An imperative language is a programming language of **statements**. Some body of code is written according to the syntax of the language and is then compiled into a program in machine code to be run. This thesis is focused on C as the imperative language of choice, in part be cause it is widespread and influential. There are others however, including other members of the C family such as C++, and separate languages like Rust. Assembly is the most basic imperative language, in which each statement is a single instruction.

A statement is a chunk of code that produces one or both of a result and side effects. A side effect is some change to the total state of the program. It generally takes the form of the change of some value stored at a known location in memory. The result if it exists is some value that is produced or "returned" as the final step of the computation of the statement.

Languages like C exist to simplify the process of writing a description of a computation. These simplifications are numerous and varied, but include several key improvements compared to writing assembly. The first is that the same C code can be compiled to run on more than one kind of machine. So details such as how many registers exist are no longer the problem of the programmer, and instead are known and dealt with during the compilation process. Next are **variables**. Variables in C are named locations to store data, abstracting away the differences between register and memory along with representing complex data structures that may not fit in

a single register. Instead of managing the access to some data in a register or in memory (including choosing if and when to move data) manually as in assembly, in C a programmer can simply indicate that the input value to a computation should be drawn from a particular variable, and leave those small details aside.

The most important difference when moving to C from assembly for our purposes is that C makes explicit the structure of a program. The flow of a program in assembly is simply some unstated relationship between labels, with some shared understanding of the state of data at the transition. In C, a **subroutine** (a small computation that is likely to be reused) has not only a name, but also an associated set of inputs and outputs. The code that calls the subroutine to use its computed value only needs to know on what the inputs, outputs, and side effects of the subroutine are, and is less concerned with how the data gets there.

C also has **types**. A type is a complicated thing in general, but in C it simply describes the layout and interpretation of some bits. Every value has a type, and variables have a type. Types indicate and restrict what sort of operations are reasonable on some piece of data. Much of the latter part of this thesis is devoted to a deeper exploration of types. Inputs (**arguments**) and outputs of subroutines are explicitly annotated with types.

What C and other imperative languages share with assembly is that they are descriptions of the action of the computer. They are basically complicated shorthand for a program in machine code, which the reader will recall is a complete description of the actions taken by the computer to perform some computation. We will see momentarily that this is only one of the ways to produce a program that performs a desired calculation.

1.4 Functional Languages

Another common way to describe a desired computation is via a **functional** language. Functional languages are a broader category of languages than imperative languages, but what they generally share is that they are less concerned with the location, representation, and source of data, and more concerned with describing the sequence of functions that produce the final result. Languages like C describe when and where to load and store from memory, how the execution of the program should progress, and generally describe actions taken by the computer. Functional languages do not describe loads and stores or explicit code flow in terms of sequential execution.

This thesis is concerned with the lambda calculus, the first functional language (aside from mathematics itself perhaps) and Haskell. As such, the description given here is meant to give a general impression of them specifically, at the cost of an imperfect description of functional languages overall. The core idea of a functional language like these two is that a description of the functions that produce the final result of a computation is the same as a description of the computation itself. Haskell is (generally) compiled into a runnable machine code program, while lambda calculus is generally for theory crafting and is not traditionally run on real machines, though it is possible to do so. In either case, the compilation or translation process makes all

the decisions about where data goes and when and how to call subroutines. The programmer simply needs to produce a description of the intended function computed by the program in terms of smaller functions. The programmer defines smaller functions (which often become the subroutines of the machine code program) which can be put together into the complete function. The smallest functions are provided by the language itself, and generally consisting of the basic operations on the fundamental data types like integers and lists of values (the implementation of which on the computer are left to the compiler).

Hopefully the reader has the impression that functional languages are more abstract descriptions of a computation. The key difference is that a functional language seeks to describe the function being computed, while imperative languages and assembly describe the actions of the computer to compute said function.

Both functional languages and imperative languages replace the low level control flow of jumps with more natural human constructions. Particularly of note are conditionals. In machine code and imperative languages, a conditional controls which parts of the code are executed. In a functional language, a conditional is a function of three inputs, and the function produces the second argument if the first is true, and the third otherwise. So for some fixed arguments a, b, c. `if` a, b, c is equivalent to b if a is true, and c otherwise. Iteration and function calling are more similar between imperative and functional languages. Iteration takes some chunk of code and runs it repeatedly, often until some condition is met. This is also expressible in a functional style, as we will see. In an imperative language, calling a function (subroutine) causes it to be run, and the result to be returned to the caller. In a functional language, the concept of "running" code doesn't really exist. Calling a function is simply applying the function to its arguments. The function along with its arguments together represent a value of the return type of the function, as the computation to compute a value and the value are equivalent in this functional setting. Regardless, special mention should be made of **recursion**, a function calling itself. This is legal and useful. In fact, it is natural to formulate iteration in terms of recursion in many functional languages. Each iteration of a loop becomes one call of the recursive function, which either returns some value to its caller or does some computation and calls itself again, often with different arguments.

Listing 1.1: Recursive Haskell Implementing GCD algorithm

```
1 gcd :: Integer -> Integer -> Integer
2 gcd a b
3   | b == 0 = a          -- Bottomed out, we have the answer
4   | b > a = gcd b a     -- Ensure a > b for all steps by swapping args
5   | otherwise =
6     gcd b r             -- With our assumptions, do one step and recurse
7   where r = a `rem` b -- The remainder of dividing a by b
```

1.5 Categories

As functional languages are generally build out of relatively few and relatively simple base constructs, their structure and power comes from their emergent properties. As a result, functional languages are generally more succinctly representable in theory and are thus the target of the majority of the theory of programming languages. One mathematical structure that sees some use in this setting is a **category**. This section gives a brief definition, limited to just the bits that are the most relevant. The primary use case within this thesis is for describing aspects of Haskell's type system, which the second half of this thesis covers in more depth. This sections simply lays out the definitions for those unfamiliar or in need of a refresher. This description is based on Messick [2007].

A category $\mathcal{C}$ has three parts.

1. A collection of **objects**. These are the elements that we move between.

2. A collection of **morphisms**. A morphism is a way to turn one object into another which respects whatever structure one might care about within the objects. Frequently they are functions that have some additional properties. Each morphism f has a domain (where it starts) and a codomain (where it ends). It is frequently useful to talk about the set of morphism between two objects. We write $\mathrm{Hom}_{\mathcal{C}}(A, B)$ for the set of morphisms from A to B in $\mathcal{C}$. We also write $f \in \mathcal{C}(A, B)$ which says that f a member of $\mathrm{Hom}_{\mathcal{C}}(A, B)$.

3. A composition law that tells us how to combine morphisms. For any two morphism $f \in \mathcal{C}(A, B)$, $g \in \mathcal{C}(B, C)$, we have $g \circ f \in \mathcal{C}(A, C)$. This law also obeys two axioms:

   (a) Composition is associative, with $h \in \mathcal{C}(C, D)$, $(h \circ g) \circ f = h \circ (g \circ f) = h \circ g \circ f$.

   (b) There are identity morphisms for each object. For all objects $C \in \mathcal{C}$, there is some $1_C \in \mathcal{C}(C, C)$ such that $1_C \circ g = g$ and $h \circ 1_C = h$.

   This can be thought of as putting the "arrows" of two functions tip to tail in the order that makes sense. The two axioms simply say that one can insert arrows that don't go anywhere without changing the resulting arrow, and that the order one combines three or more arrows doesn't effect the result.

A canonical example of a category is the category of sets, **Set**. The objects are all sets and the morphisms are functions between sets under standard function composition.

We say some $f \in \mathcal{C}(A, B)$ is an **isomorphism** if there exists some $g \in \mathcal{C}(B, A)$ such that $f \circ g = 1_B$ and $g \circ f = 1_A$. Namely g takes every object back to where it started before f was applied and vice versa. If such such f, g exist, we call A and B **isomorphic**. Isomorphic objects are somewhat interchangeable in a category, as any morphism involving one can become a morphism involving the other by applying the

f or g before or after as appropriate. Informally, one can always get from one to the other.

A **functor** F is a map (read function) from one category $\mathcal{C}$ to another $\mathcal{D}$. It produces an object in $\mathcal{D}$ for each in $\mathcal{C}$ (though they may not be distinct) and a morphism in $\mathcal{D}$ for each in $\mathcal{C}$. Furthermore it is required to obey the following laws.

1. $F(1_C) = 1_{F(C)}$. This says that the identity on an element in $\mathcal{C}$ will become the identity on the corresponding element in $\mathcal{D}$.

2. If $h = g \circ f$ in $\mathcal{C}$, then $F(h) = F(g) \circ F(f)$. This says that it doesn't matter if one applies the functor and then composes or composes and then applies the functor, they are equivalent. Equivalently, the diagram 1.8 commutes. That is, all paths from A to γ are equivalent.

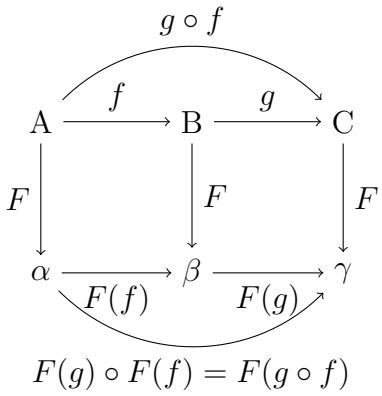


Figure 1.8: Functor F between $\mathcal{C}$ and $\mathcal{D}$

The other categorical structure we are interested in are **monads**. This definition is based on one by Mulry [1998]. Let C be the category of sets. Fix some monoid M. As a reminder a **monoid** is some set M together with a associative operation (written as multiplication and thus generally lacking a symbol) and a identified element $e \in M$ that is the identity for the operation. Thus for $a, b, c \in M$ we have

$$abc = (ab)c = a(bc)$$
$$em = m = me$$

as we would expect.

Let H be the endofunctor (functor from C to C) with $H(A) = M \times A$. A monad on C is a triple of (H, μ, η), the endofunctor and two natural transformations. One is

$$\eta_A : A \to M \times A$$
$$a \mapsto (e, a)$$

and the other is

$$\mu_A : M \times (M \times A) \to M \times A$$
$$(m, (n, a)) \mapsto (mn, a)$$

Note that η produces a natural pair for any $a \in A$ by pairing it with the most "inoffensive" element in M, namely the identity which will make no difference when multiplied. Note that η_A takes an element of A to one of $H(A)$. It is the simplest way to "lift" an element of A. The understanding of μ_A that is important to us is that μ_A goes from $H(H(A))$ to $H(A)$ in the way that is natural for M. Thus μ is morally a "flattening" operation of some kind.

Using η alone allows the full categorical structure of C to be present in the richer setting of the monad, as for any morphism in C, there is a new morphism given by post-composing with η, which adds no structure, but results in a morphism that operates in the monadic space rather than the underlying one.

The critical thing for us to understand about monads is how to use μ. The most useful feature of μ for our purposes is that one can use it to define a kind of composition. If one has two functions of the appropriate shape, $f : A \to M \times B$ and $g : B \to M \times C$ then one can compose them in a natural fashion. We see we can define

$$\circ_{(M,C)} : (B \to M \times C) \times (A \to M \times B) \to (A \to M \times C)$$
$$(g, f) \mapsto \mu(f_M, (g \circ f_B))$$

Where f_M is the the function $A \to M$ by applying f and taking the first element (in M). Similarly f_B is applying f and taking the second element (in B). Note that since M is a monoid and $\circ$ is natural function composition, this new enriched composition is associative. We will see that this gives us a natural sequencing operation. If we have some sequence of functions $f_1, f_2, \ldots, f_n$ (each $S_{i-1} \to M \times S_i$) through a sequence of sets S, then this composition operator gives us a function $f : S_0 \to M \times S_n$ that captures both the standard composition of f_is if one ignores the monoid element, and also a trace of the monoid elements in the form of some monoid element that is the multiplication of the monoid elements produced by each f_i in order.

Thinking of η and μ as natural transformations $\eta : id \to H$ and $\mu : H^2 \to H$, we also have that a monad obeys

$$\mu \circ \eta_H = id_H = \mu \circ H_\eta$$
$$\mu \circ \mu_H = \mu \circ H\mu$$

The exact details are not critical here, but these laws basically ensure what one might expect, namely that packing and unpacking in any order shouldn't change the underlying value.

Formalism aside, monads allow one to add auxiliary associative structure to a category. Categories are important for us because they are critical to Haskell, one of our central programming languages of interest. The chapter on Haskell goes into more detail about why categories are necessary, but in short they allow Haskell to have a consistent and powerful type system while representing effects (notably input and output) that are not naturally represented by standard types alone.

Chapter 2

P-Code

> All problems in computer science can be solved by another level of indirection.
>
> David Wheeler

The previous chapter gives a model of computation. The model of computation together with the precise details of how the addressable range, register number, register size, and other details are selected and cemented together with an instruction encoding for a given computer. These choices taken together form a **Instruction Set Architecture**, or ISA. Naturally for any decision made during the ISA creation process, there are other options, and often some other ISA that has made the other choice. So while the choice of ISA is arbitrary, since for the most part it does not effect what a program can and cannot accomplish, it does make a significant difference in terms of what some computation might look like at the instruction level.

Most programs are written in a high level language that allows a program to be created without knowledge of any specific ISA, and in fact the same program can be compiled for many ISAs. Thus tools that operate on source code have applicability to many ISAs for free, since the semantics of the language hide all of those details. However, debugging, some security tests, and other problems of correctness and trust are often interested in programs at the instruction level. Tools in those areas are then forced to specialize or be general over several ISAs.

One such tool is Ghidra Directorate [2023], a tool designed by the National Security Agency and later open sourced. Ghidra is a tool that takes a compiled binary and attempts to build a readable source code for it. An explanation of how this process works is out of the scope of this thesis, but in general this process is called decompilation. Decompilation is the task of creating (C) source code that compiles to the given binary program. This process is challenging, complex, and immensely useful for certain security applications, among other trust problems in Computer Science. Another selling point of Ghidra is its near universal knowledge of ISAs. Naturally this poses a problem for the developers of Ghidra, as the nature of decompilation makes it incredibly dependent on choice of ISA. It would be a fool's errand to attempt to solve the difficult problem of decompilation for each ISA, particularly if one requires

that they produce similar looking source code.

P-Code is the solution Ghidra takes to this problem. P-Code is an extremely general ISA that does not correspond to any piece of extant hardware. Its purpose is to be simple and small enough to be easily analyzable, but flexible enough to translate into from other ISAs. Ghidra contains two translation layers between the initial loading of a binary file and the primary decompilation process. The first is to convert the given ISA to **Low P-Code**. This step consists of, for each instruction in the input ISA, converting it into one or more P-Code instructions, such that that they perform the exact same actions as the original instruction. For simple instructions, that may be a one to one conversion to the equivalent P-Code instruction (for instance summing two registers and placing the output in another register), but for others it may be far more involved. (Some complex ISAs, such as `x86_64` might do a number of things in one instruction, such as the return instruction that returns to the caller indicated on the stack, and pops some number of other items off the stack.)

Once a program is converted to an equivalent program in P-Code, the second translation level massages it from Low to **High P-Code**. High P-Code is almost the same, but is further abstracted away from the hardware. It contains explicit information about control flow and other higher-level program information.

2.1 Technical Details

This section contains a brief overview of the aspects of P-Code that differ from standard ISAs and are of interest to us based on the reference manual Agency [2019].

2.1.1 Address Spaces and VarNodes

One of the primary differences between ISAs is the question of which instructions can touch memory, and which instructions are limited to operating on registers. For instance, RISC-V only allows touching memory through explicit loads and stores, but `x86_64` allows some instructions, like integer arithmetic, to take one input from memory directly. These decisions are borne from balancing speed and hardware complexity, among other factors. P-Code is meant to be general over all ISAs, so we need some way to encode instructions that might touch memory more than once. We could break such an instruction down into explicit loads and stores around a register-only operation, like we might do in RISC-V, but that is inflexible and inefficient. This is particularly true given that P-Code doesn't have the constraint of having to be supported directly in hardware.

Thus in the pursuit of generality, P-Code leans heavily on the concept of an **address space**. An address space is a generalization of memory, an indexed sequence of bytes. Note that this is not to be confused with the operating systems term address space; here it is just an abstract sequence of bytes. For a single byte, its index is the byte's address, which is unique. An address space is composed of a name, a number of bytes that it contains, and information about how to lay out values that are more than a byte long (endianness).

Programs in P-Code generally have three main address spaces: The `ram` address space, which corresponds to memory as the standard assembly programmer knows it, the `register` address space, which generalizes the set of registers the ISA supports, and a `constant` address space, which has constant values that might be used during the P-Code translation, including constants encoded in the original instructions, like the immediate offsets in RISC-V operations. There is also sometimes a `temporary` address space, which can be useful in the translation, but doesn't correspond to any part of the original program. The `register` address space is of particular interest, since it differs in arrangement from most ISAs. For instance, in RISC-V, registers are disjoint locations that do not share any sequential relationship, other than the fact that they are numbered. However, in P-Code, the second four byte register is simply the next four byte sequence in the `register` address space after the first four bytes. This is a strictly more general view of registers than that of RISC-V, but it allows for a uniformity between memory and registers, and also matches some ISAs, in which a large value is split and stored in two registers, often next to one another.

To talk about a location, memory or register, P-Code uses **VarNodes**. A VarNode is simply an address space, an offset therein, and a length. This has a natural interpretation in the case of the `ram` address space, as it is designed to mimic the way memory normally works. In the case of a register, the VarNode simply indicates a region of the `register` address space according to the mapping of registers to regions of the `register` address space.

The other aspect worth mentioning is the length. For a normal ISA, there is generally one primary granularity of data (four bytes in the case of the base RISC-V-32 ISA), though they also generally support operations on smaller units (16 bit integers for example). P-Code is even more flexible, by allowing each operation to act on arbitrary lengths, depending on its input and output lengths.

2.1.2 Instructions

A brief aside should be taken to talk about instruction order in P-Code. In a normal ISA, instructions are simply executed in program order, barring control flow altering instructions like branches or jumps. However, there is a slight caveat with P-Code instructions, arising from the fact that there can be more than one P-Code instruction in the translation of a single machine (input ISA) instruction. Thus P-Code instruction sequencing abides by the following rule: The next P-Code instruction to be executed is the next instruction in the translation of the current machine instruction, and if there are no more P-Code instructions in this machine instruction, then the next sequential machine instruction is begun, starting with the first P-Code instruction in its translation. Thus straight line computations are as expected: each machine instruction in order, and each sub-unit of translation in order.

The issue of branches and other non-linear control flow is handled similarly. A P-Code branch can either branch within the scope of the current machine instruction, called a P-Code-relative branch, or between machine instructions. In the latter case, P-Code execution begins with the first P-Code instruction of the targeted machine instruction's translation. It is impossible to branch directly to a P-Code instruction

in another machine instruction that is not the first one.

All P-Code instructions use VarNodes for inputs and outputs, and are thus general over registers, memory, and any other address space that might exist for a program and piece of hardware. There will not be an exhaustive list of P-Code instructions, as they are largely the same as any simple ISA, up to the flexibility from using VarNodes.

P-Code contains all the elementary integer operations of a standard ISA, along with instructions to check for overflows and underflows associated with fixed precision addition and subtraction operations. Boolean logic operations, floating point operations, and conversions between floating point and integer encodings are also represented. Conditions of the sort found in the conditional branches of RISC-V are represented as explicit operations that take integers or floating point values and produce a boolean value. Conditional branches take an extra boolean valued VarNode to switch on, rather than having the condition be internal to the instruction. No differentiation is made between branches and jumps, and no registers are saved or manipulated during those operations. If one wants to save the jumped-from location to the stack as in a subroutine call, it must be done explicitly before the jump occurs. The branch operations simply alter the next instruction to be executed. P-Code supports both branches to constant locations (VarNodes with a constant address space) and branches/jumps to a location indicated by a non-constant (often register) VarNode. P-Code also calls these branches, though in some ISAs they would be called jumps. P-Code only allows indirect branches like these between machine addresses, while constant branches can be P-Code relative. Indirect branches are also not allowed to change address spaces, while constant branches can. The exact details beyond this short description will be included in later explanations if they become relevant.

2.2 Pitfalls

Unfortunately, while P-Code is accessible directly through Ghidra, and in theory available to the public, it is not really part of the forefront of Ghidra's open source interface. This section briefly covers some of the difficulties and subtleties of working with P-Code.

2.2.1 High and Low P-Code

An unintuitive aspect of P-Code is that the core P-Code documentation and Ghidra implementation actually describe two languages. These languages are largely overlapping and lack canonical names, but we will be following several other academic works and be referring to them as **High P-Code** and **Low P-Code**. Other names for Low P-Code include Raw P-Code. Generally when unqualified, "P-Code" refers to High P-Code, but this work is exclusively concerned with Low P-Code.

The first few steps that Ghidra takes on loading a binary illustrate the difference. Initially the binary is read and the header meta-data is inspected. Once the target ISA in question is determined, each target ISA instruction is converted to one or more Low P-Code instructions. These Low P-Code instructions capture exactly and

completely the effects of the target ISA instruction, including any CPU flags or side effects, but do no analysis beyond that. Once all the instructions in the target binary have been converted to Low P-Code, a number of transformations are applied to the Low P-Code to produce High P-Code, which is used by the rest of the Ghidra stack. There are two main transformations that are of interest to us. These are the identifications of functions and the production of phi-nodes.

The former is straightforward: Ghidra identifies where in the instruction stream a function seems to begin. Ghidra targets binaries produced by C and C-like languages, so this process is mostly just a task of collecting the targets of target ISA instructions intended for sub-routine calls (`JALR` and its ilk). In P-Code there are all merged into a single `CALL` instruction that acts like a branch, and may be combined with other Low P-Code instructions to perform things like saving the caller program counter like many ISAs do.

Briefly, a phi-node occurs whenever a sequence of instructions is reachable from more than one place. A contiguous sequence of instructions that is atomic with respect to control flow is called a **block**. By that we mean that when executing the program, either each instruction is executed in the block exactly once in order for each time the block is reached, or the block is not reached. A block may only contain control flow instructions as the final instruction (called the terminating instruction).

Consider a block that is reachable by exactly two blocks. The first predecessor block writes 1 to `x1`. The second writes 2. If an operation in the current block wants to use the value provided by the predecessor block, it would simply use `x1`. However the reader will note that when concerned with the program in general, and not with a specific execution of the program, it is not clear what the value of `x1` is. It's not 1 or 2, since its not immediately clear what the predecessor block is. However the value is also not completely general, since at the start of the block it must be either 1 or 2, and no other value. A phi-node encodes this constrained ambiguity. A **phi-node** is a mapping from the set of possible prior predecessor blocks to the set of a values for a specific data location (generally a register). We will not provide a formal definition for phi-nodes, since we do not use them directly, and their exact formulation depends on the context the are used in. In High P-Code, they represent registers who's contents change depending on the predecessor block, or any block between the current block and its immediate dominator. High P-Code promises that any data locations that can vary depending on control flow path are captured in a phi-node. The phi-node encapsulates the information about the origin of the data, and the rest of the block can simply use the data.

The creation of phi-nodes is of lesser interest. This is because we do not rely on them in our work. The full discussion of why what is the cases is to follow 4.3.5. Phi-nodes are mentioned here because they are relevant later in terms of understanding static analysis. Typically phi-nodes are utilized in Singe Static Analysis 3.1.1, but Ghidra uses them without the rest of the structure of SSA forms.

2.2.2 Specification

Another difficult aspect of working with P-Code is the ambiguity surrounding the core pseudo-ISA. While there is a general specification in prose Agency [2019], P-Code lacks a centralized formal specification. A formal specification allows people working with P-Code (inside or outside Ghidra) to know with certainty what is and is not legal or supported by P-Code.

Prior work outside of the Ghidra developer circle has included a formal specification for High P-Code Naus et al. [2023]. That paper illustrates the importance of a formal specification, since some small changes had to be made to the form of some P-Code operations (namely the phi-node merges) to be consistently expressible. In addition, the authors had to confirm several aspects of P-Code's behavior through experimentation.

2.3 Extracting P-Code

At present, Ghidra does not provide a natural way to extract P-Code from an analyzed binary. While it is simple enough to view the P-Code representation of a region of code inside the Ghidra application, it is not currently supported to export that data for external use. To achieve this, we extended Ghidra with a plug-in that writes each function to a file with Ghidra's name for the function and the P-Code associated with it in order. This plug-in is based on the existing plug-ins that do similar things, notably niconaus [2022] that dumps High P-Code and an example plug-in HackOvert [2023] that dumps Low P-Code for a single function. Our version differs in that it dumps the P-Code for every function, and it dumps specifically Low P-Code Ulmer [2023b]. The exact format of the dump is not important, but mirrors the expected structure of our interpretation code, which produces verbose but unambiguous representations.

2.4 Example Translation

Listing 2.1: Example C function

```
1
2 int main () {
3   register int i = 0;
4   while (1) {
5     ++i;
6   }
7   return 0;
8 }
```

Listing 2.2: Abreviated Example C Translation

```
1 FUNCTION          main
2 0x000101da, 0x0, COPY ("const",0xffffffffffffff0,0x8), ("unique",0xe80,0x8)
3 0x000101da, 0x1, INT_ADD ("register",0x2010,0x8), ("unique",0xe80,0x8), ("register",0x2010,0x8)
4 0x000101dc, 0x0, COPY ("const",0x8,0x8), ("unique",0x1600,0x8)
```

```
5  0x000101dc, 0x1, INT_ADD ("unique",0x1600,0x8), ("register",0x2010,0x8), ("unique",0x16a80,0x8)
6  0x000101dc, 0x2, STORE ("const",0x1b1,0x8), ("unique",0x16a80,0x8), ("register",0x2008,0x8)
7  0x000101de, 0x0, COPY ("const",0x0,0x8), ("unique",0x1600,0x8)
8  0x000101de, 0x1, INT_ADD ("unique",0x1600,0x8), ("register",0x2010,0x8), ("unique",0x16a80,0x8)
9  0x000101de, 0x2, STORE ("const",0x1b1,0x8), ("unique",0x16a80,0x8), ("register",0x2040,0x8)
10 0x000101e0, 0x0, COPY ("const",0x10,0x8), ("unique",0x1280,0x8)
11 0x000101e0, 0x1, INT_ADD ("register",0x2010,0x8), ("unique",0x1280,0x8), ("register",0x2040,0x8)
12 0x000101e2, 0x0, COPY ("const",0x0,0x8), ("unique",0xe80,0x8)
13 0x000101e2, 0x1, COPY ("unique",0xe80,0x8), ("register",0x2008,0x8)
14 0x000101e4, 0x0, COPY ("const",0x1,0x8), ("unique",0x780,0x8)
15 0x000101e4, 0x1, INT_ADD ("register",0x2008,0x8), ("unique",0x780,0x8), ("unique",0xdc80,0x8)
16 0x000101e4, 0x2, SUBPIECE ("unique",0xdc80,0x8), ("const",0x0,0x4), ("unique",0xdd00,0x4)
17 0x000101e4, 0x3, INT_SEXT ("unique",0xdd00,0x4), ("register",0x2078,0x8)
18 0x000101e8, 0x0, SUBPIECE ("register",0x2078,0x8), ("const",0x0,0x4), ("unique",0xde80,0x4)
19 0x000101e8, 0x1, INT_SEXT ("unique",0xde80,0x4), ("register",0x2008,0x8)
20 0x000101ec, 0x0, BRANCH ("ram",0x101e4,0x8)
```

Chapter 3

Symbolic Analysis

> The last bug isn't fixed until the last user is dead.
>
> Sidney Markowitz

Now that we have some idea of what a computer can do and how to represent a program binary, we can return to our high level goal, namely checking the correctness of a program against a model.

This chapter presents an account of **static analysis**, specifically **symbolic analysis**. In addition, it attempts to provide motivation for why symbolic analysis might be desirable and some of the challenges of static analysis in general based on Schwartzback and Dockins et al. [2016]. This account is greatly simplified and intended primarily to motivate this project rather than to give a rigorous formulation of the problem. A more formal account based on an encoding in lattices can be found in Schwartzback.

A programmer is generally interested in computing some value according to some inputs. Naturally they want their code to compute the output correctly, and further correctly for all possible inputs. If one could convert the program to a mathematical model that captures the computation exactly, then the programmer could confirm that the program was implemented correctly. Symbolic analysis is the process of building that model and attempting to answer related questions.

A critical distinction between a mathematical expression of a value in terms of some inputs and a description of the computation that produces one such value given inputs is the notion of concreteness. In mathematics, given some term a that is known to be an integer, we can do all the same things on it that we could do on any specific integer. In short, we can manipulate the symbol a and the symbol 5 in an identical matter, because the operations such as addition are fundamentally abstract. In this case, ISA level descriptions of computation act somewhat similarly. They can manipulate a register `xn` in the same way that they do a literal value k, for instance a 5. However where this falls apart is the notion of mutability, or equivalently register reuse.

Since each machine instruction only knows how to act on inputs that are registers or literal values, if we wanted to represent a two step computation such as $(a+5)\times 2$,

then we must first place the intermediate value of $a + 5$ somewhere (i.e. a register), and then use it. This is necessary, because the `MUL` instruction has no way to represent a nested computation like $a + 5$. This alone is not a problem, as intermediate values can simply be stashed in registers and passed as an input. If one has the intermediate computation $a + 5$ in `x2`, then the `MUL` instruction can simply use `x2` and 2 as its operands, and all is well.

Unfortunately there are not an infinite number of registers, and so registers must be reused. This poses an issue to our translation, as something like $\texttt{x1} \leftarrow \texttt{x1} + 1$ is a perfectly valid step of computation, but is meaningless in the naive attempt to translate into a mathematical expression.

In short we need to perform some transformations to our description of computation before we can lift it into a mathematical expression for the computed value in terms of the inputs.

3.1 Atoms, Composition, and Straightline Computations

Given a concrete program that computes some function, we want to create a mathematical object, likely an expression in the inputs, that captures the **semantic meaning** of the steps performed by program. By semantic meaning, we are speaking specifically about the information captured in a known representation (ie. the computed number instead of a description of the location and layout of some bits). We want our resulting object to speak about numbers, rather than being a precise but unhelpful description of bit manipulations. To that end, our translation process must account for the conventions of representation. One difficulty of translating is that some manipulations are fundamental to the bit-representation but may not be fundamental in the corresponding represented domain, e.g. the integers. We will see an example in the next section.

The rest of this chapter will be concerned with the concepts of translating RISC-V assembly into formal mathematical expressions. This process is similar to those for other assembly variants and bears some resemblance to the equivalent process for a higher level language. This formulation is original and inspired by several functional languages including Lisp and Haskell along with the standard formulation of SSA forms.

3.1.1 Eliminating Mutability

The first issue that arises is that computation in assembly is fundamentally about manipulating some finite set of reusable registers. However, mathematical expressions have no concept of location. A number simply exists, whether it is the final result or an intermediate step. For example, a common program step is $x = x + 1$, which is a natural operation for a machine with registers, but a nonsensical statement about mathematical expressions.

The primary way this is handled is using a normalized representation called **Static**

Single Assignment. SSA form represents a computation in which each register can be written to exactly once, each register must be defined (written to) before it can be used, and there are arbitrarily many registers. SSA form in its modern incarnation is the product of Cytron et al. Cytron et al. [1991], which along with its predecessors presents how to convert into SSA form and some of the benefits.

Static Single Assignment makes the dependencies between intermediate values and the data flow of the computation to be explicit. For RISC-V assembly specifically, the translation process is primarily composed of splitting registers into versions, with each assignment of the register becoming a new version, and each read from the register becoming a read from the version most recently defined in program order. This is similar to the mathematical convention of appending apostrophes to signify iterated versions of a value. The original code might say $\texttt{x1} \leftarrow \texttt{x1} + 1$, our SSA form might be $\texttt{x1}_2 \leftarrow \texttt{x1}_1 + 1$, and the mathematical version might look like $x1'' \stackrel{def}{=} x1' + 1$, for instance. The details of translating memory and control flow will be discussed in later sections.

This largely mechanical transformation is already a large step forward. We now have a **intermediate representation** (IR) that captures our interest in data flow rather than data location. Data is immutable, but can be used to compute new pieces of data.

3.1.2 Atoms

The most indivisible computations that a RISC-V machine can do are a single instruction, i.e. `ADD` or `XOR`. We recall that these operations can take either registers, R, as inputs or immediate constant values, C. Thus we can characterize the fundamental type of data being the disjoint union

$$D \stackrel{def}{=} R \sqcup C$$

A single data atom is either a register or a constant. Then a very natural formulation for a function atom is

$$F \stackrel{def}{=} \{f \mid f{:}D \times D \to R\}$$

the set of binary functions on the domain of data atoms. Note that all $R \subsetneq D$, so this function is strictly contained in the domain of data atoms.

From these definitions it is fairly clear that the data manipulation instructions of RISC-V are function atoms, and the registers and immediates are data atoms. A dubious reader can justify to themselves that all of the data manipulation instructions presented in the RISC-V reference in the appendix A are all easily represented mathematically given some knowledge of bit-level representation conventions.

Note that this is a strictly more general representation of the computations possible in a single instruction of RISC-V, since it would be an error to write instructions that take two constants as input. However, since RISC-V is a strict subset, we can convert into this IR, so long as we aren't concerned with converting back. The motivation for this IR is to eliminate the difference between immediates and registers,

since that is an aspect of data location which we don't care about, and in SSA form registers are also immutable, so the difference is largely arbitrary.

Thus with data atoms and function atoms, we can express a single instruction's worth of computation on a RISC-V machine in what we will call the FD-atom IR. The next question is obvious: What about more than one?

3.1.3 Composition of Atoms

Consider the following RISC-V excerpt:

Listing 3.1: Multi-instruction computation

```
1 add x2, x1, 2   ; x2 <- x1 + 2
2 sub x4, x2, x3  ; x4 <- x2 - x3
```

In prose, this excerpt adds 2 to the contents of `x1`, stores the result in `x2`, subtracts the contents of `x3`, and leaves the results in `x4`. Note that `x2` contains an intermediate value after this excerpt. Given that there is only one write per register involved here, and assuming that `x1` and `x3` were defined prior to this excerpt, then no actions are necessary to make this a SSA form.

In the IR we are constructing, we can express the first instruction as

$$\texttt{x2} \leftarrow \texttt{x1} + 2$$

where 2 is a constant, and the second as

$$\texttt{x4} \leftarrow \texttt{x2} - \texttt{x3}$$

However if we are concerned with the final value, and not the location or intermediate computations, we might write instead

$$\texttt{x4} \leftarrow (\texttt{x1} + 2) - \texttt{x3}$$

This composition of function atoms allows for a very intuitive representation for computations that result in a single computed value. Furthermore, this formulation captures the high level pattern of writing or generating assembly, namely the breaking of large computations into increasingly simpler ones, until they can be represented trivially i.e. in a single instruction.

Consider taking the determinant of a 2 by 2 matrix of integers. Anybody who has studied linear algebra will you that for a matrix

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

the determinate is $ad - bc$. Now consider a plausible subroutine that does that computation, taking inputs in `a0-a3` and placing the result in `a4`. The reader can assume that no registers are aliases of one another.

Listing 3.2: 2 by 2 determinant

```
1 mul t0, a0, a3 ; t0 <- a0 * a3
2 mul t1, a1, a2 ; t1 <- a1 * a2
3 sub a4, t0, t1 ; a4 <- t0 - t1
```

Again we only care about the final value in `a4`. Our IR (with renaming to match) gives us $(ad) - (bc)$, exactly the semantic calculation, stripped of the messy details of which register the data resides in, and where the intermediate values came from.

3.1.4 Straightline Computations and Side Effects

This formulation alone, even without any further additions, is quite powerful in terms of what RISC-V computations it can express. As presented so far, the FD-atom representation and its associated translation process can express any RISC-V machine computation that does not feature any control-flow operations. Computations that do not require control flow are called **straightline computations**. And the FD IR gives us a way to take a straightline computation and get an expression for the output with symbolic names for the inputs.

One further generalization can be made without adjusting the process. So far our translation efforts have been towards finding an expression representing a singular output of a computation. However it is reasonable to ask about computations that produce more than one output, or have side effects other than the primary output. An example could be a subroutine that takes two numbers, divides the first by the second, and returns the quotient as an output, and leaves the remainder in the location of the first input. These **side effects** of a computation can be thought about in two ways. The first is to think about them as secondary effects. In our determinant example 3.2, the contents of `t0` and `t1` are changed, but they don't contain the information we care about, or at least not the complete information. The other method is to roll the side effects into the final output. Again for the determinant, one could instead say that the subroutine has the output of the tuple $(ad - bc, ad, bc)$ in the locations `a4`, `t0`, `t1`. In this view, while parts of the output may appear less than useful, the computation has no side effects, because they are included explicitly as the output. While this is not generally the mindset of a programmer using side effects, this method of framing can be applied universally.

A related property of being side effect-free is for a function or subroutine to be **pure**. A pure function is side effect-free and has output dependent solely on the immediate arguments to the function. This is in contrast to a function that might consult a data location that was not explicitly supplied at the call site, e.g. a global variable in many programming languages. Pure functions have referential transparency; replacing the call with the return value does not effect the computation.

To capture the side effects of a subroutine in the FD-atom IR, we can simply ask for a FD expression for each data location that is written to during the course of the subroutine. This is in essence the same as the tuple method. Often the resulting expressions for the side effects may be subexpressions in the primary output, for example the side effect on `t0` is ab, which is a subexpression of the output in `a4`.

The full generalization of the FD IR takes any straightline excerpt and produces an expression for each register, each with a free variable for every register, regardless of if it appears in the excerpt. Thus if $\vec{r}$ is the set of register states, where each register takes a value from the same domain S, the full IR for an excerpt is a function from a complete register state (tuple of length $|R|$), to a new register state.

$$f{:}S^{|R|} \to S^{|R|}$$

such that $f(\vec{r})$ is the set of register states after the excerpt is run with initial register values $\vec{r}$. A convenient shorthand notation is f_i, which takes the initial complete set of register states to the resulting state of register i: $f_i{:}S^{|R|} \to S$.

This full generalization is useful because it has the same shape as a mathematical object regardless of the contents of the excerpt, and is still completely correct even if one doesn't know the location of the primary output of the excerpt.

3.2 Control Flow

We have given a nice framework for discussing straightline computations. The natural next step is to attempt to model the ways in which straightline computations are combined: conditionals, subroutines, iteration, and recursion.

Fundamentally, both recursion and iteration are concerned with repetition. It is well known that recursion and iteration are equivalent, and we can convert one to the other simply. However there are still distinctions to be made, namely between three classes of repetition: fixed number iteration, primitive recursion, and general repetition. Fixed number iteration is the simplest. It contains any repetition that occurs a fixed number of times that is known at compile time. The same number of loops or recursive subcalls will happen regardless of input. Note that this is not fundamentally different, as fixed number iteration can be unrolled into a straightline computation by simply duplicating the contents of the loop the appropriate number of times.

Primitive recursion contains instances of repetition where the number of iterations or the depth of the recursion are known before the first repetition begins. This term is generally phrased in terms of recursion depth, but is equivalent to the most common form of `for` loops in many programming languages. Note that these kinds of repetition cannot be unrolled into a straightline computation, as the "length" of the straight line is not known before the program is run. The core difference between this and the first kind of repetition is that the number of iterations differs between inputs, but for each input it is known before it begins.

Finally general repetition is the most broad category. It contains the natural extension of the above, namely cyclic computations in which the number of repetitions or cycles is not known before the process is begun. This can be conceptualized as a `while` loop with a nontrivial condition (ie. one that can't be converted into a traditional for loop in terms of an indexing variable), or alternatively a flow chart containing a cycle or loop.

The third, most general version of iteration is behind the famous halting problem, which in layman's terms says that knowing if a program eventually completes is as hard as running it. (And therefore may not finish in finite time, if the program doesn't halt.) This in turn means that even categorizing an instance of repetition as expressible in which class is also undecidable in general. Though for many practical examples, it is not hard.

Despite the impossibility result lurking in the background, one can attempt to chip away at practical cases of representing effects of cyclic processes.

3.2.1 Conditionals and Subroutines

From this point on, assume that all functions detailing register effects take as arguments and produce as output full register sets in the manner described above.

Consider a two segments of code that have some register effects $E_1(\vec{r})$ and $E_2(\vec{r})$. Consider code that executes the first block if `x1` is zero, and the second otherwise. This is not a straightline computation, but a natural expression presents itself if we allow ourselves piecewise functions in our symbolic representation. Then we can write that the global effect of this combined conditional segment is

$$E'(\vec{r}) = \begin{cases} E_1(\vec{r}) & \text{if } \texttt{x1} = 0 \\ E_2(\vec{r}) & \text{if } \texttt{x1} \neq 0 \end{cases}$$

While this may seem undeserving of mention, a subtle difference exists. The conditional in the code directs the execution for a single run of the program. The conditional in the expression above is a single expression that is true for all inputs.

Consider applying this to a absolute value example.

Listing 3.3: RISC-V Implementing Absolute Value

```
1  # takes a integer in x1 and produces its absolute value in x2. x0
2  # always contains the value zero
3  abs:
4    blt x1, x0, negative     # if x1 < x0, go to "negative", else continue
5    mov x2, x1               # assign x2 = x1
6    j exit                   # go to "exit"
7  negative:
8    mul x2, x1, -1               # assign x2 = x1 * -1
9  exit:
10   return
```

We see naturally that the expression we want is

$$\text{abs} = \begin{cases} \vec{r}[\texttt{x2} \leftarrow \texttt{x1}] & \text{if } \texttt{x1} \geq 0 \\ \vec{r}[\texttt{x2} \leftarrow -\texttt{x1}] & \text{if } \texttt{x1} < 0 \end{cases}$$

Where $\vec{r}[a \leftarrow b]$ means $\vec{r}$ with a replaced with b. While this is a simple example, note that effects of high complexity affecting many registers could be placed in each arm instead of these simple assignments.

A similarly natural solution exists for subroutine calls. Putting aside questions of diverging programs, optimized tail calls, and other non-standard control flow between

subroutines, if a code segment calls out to another code segment, then the majority of the time it will return to the calling location. Under this model, if we have some effect $E_s(\vec{r})$ for the subroutine's changes to the registers, we can simply substitute the call in the code for a symbolic statement that applies those effects. This requires some careful handling that is out of the scope of this thesis, but in short it introduces the question of evaluation order (which makes a difference).

One should note that the above strategy requires an expression for the callee function to already exist. Thus it is unsuitable by itself for recursive functions (or any cyclic call tree).

Now our representable languages are composed of straightline segments connected by conditionals and subroutines, both of which can have arbitrarily nested contents, subject to the rules of no iteration and no cycles.

3.2.2 Attempting Iteration

As detailed above, fixed number iteration presents no issue for symbolic representation. *Unrolling* converts loops of that variety into straightline computations with equivalent effects. Not only is this process comparatively simple, it is well documented and studied, as it is a common optimization technique.

Primitive recursion and its iterative counterpart are difficult to capture in full generality, and are a core subject of research in static analysis. How to handle them in general is out of the scope of this thesis, but much of the attempt is centered around **invariants**. An invariant is a property about the data involved in a loop or recursive call. Armed with the symbolic effects of the body of a loop, one can often produce an invariant such that the property holds at the beginning of the loop and after each iteration. This together with the variable of iteration (which contains how many loops have been performed so far) can often produce a full symbolic effect for the entire loop. The recursive version is similar and bears a striking similarity to mathematical induction.

General iteration is difficult, but the general tactic is to fake it. Some large value k is chosen, and the iterative or recursive code is transformed to run for at most k iterations. If it would go beyond k, the analysis has failed. The code may still exit before k iterations have occurred (or equivalently the remaining iterations have no effect). After being transformed in this manner, the problem can now be solved with the tools for primitive recursion. For a suitable choice of k this is often good enough Barrett [2021]. Note that the undecidability result states that it is impossible to have a fully general version that terminates for all programs.

Chapter 4

Our Work

I met a traveller from an antique land,
Who said—"Two vast and trunkless legs of stone
Stand in the desert. . . . Near them, on the sand,
Half sunk a shattered visage lies, whose frown,
And wrinkled lip, and sneer of cold command,
Tell that its sculptor well those passions read
Which yet survive, stamped on these lifeless things,
The hand that mocked them, and the heart that fed;
And on the pedestal, these words appear:
My name is Ozymandias, King of Kings;
Look on my Works, ye Mighty, and despair!
Nothing beside remains. Round the decay
Of that colossal Wreck, boundless and bare
The lone and level sands stretch far away."

Percy Shelly

At this point, the reader has a fairly complete view of what the purpose of this thesis is. The purpose of this chapter is to cover the interesting high level details of how we went about achieving the goal of bringing symbolic analysis to binaries. After this chapter, the reader will have a near complete view of what our code does. The remaining chapters following this one detail some sources of small scale complexity that are of particular interest.

This chapter presents an account of the state of the problem and where our work fits in. In short, our work brings arbitrary binary programs from compiled C-like languages into scope for symbolic analysis when lacking the original source code. In simpler terms, extending the ability to ask "What does this program, or program fragment, do mathematically?" to programs that have already been compiled.

4.1 The Lay of the Land

The most important player to introduce is the library Crucible. Crucible is a library written in the programming language Haskell by a team at Galois Inc. The purpose of Crucible is to provide a language of symbolic computation, and methods of converting standard programming language variants into the symbolic language.

Given a program in Crucible's representation, one can then perform symbolic queries about the program via one of Crucible's front-ends, or tools built on top of Crucible. For instance, the symbolic simulator lets one provide some assumptions about the input to a function, and expectations about the output, and have a confirmation that the final expectations are met for all valid inputs. These queries return either a confirmation that the expectations hold under the assumptions (which represents a mathematical proof that the given implication between the inputs and outputs of the function holds), or provides a concrete counter-example.

The natural question is how to produce the Crucible representation? The Crucible library also includes a number of back-ends that can convert some real-world intermediate representations into the Crucible representation. Examples include converting programs expressed in LLVM, Mir (the Rust compiler's IR), Web Assembly, Go-lang, and Java Virtual Machine.

In addition, formulating queries can be cumbersome, so front-ends exist to integrate Crucible more naturally into the development process. One such front-end is Crux, which wraps Crucible and presents itself as a library in the language of choice. For instance, in Rust, Crux-mir allows one to express the assumptions and expectations around a function directly in the source code. This is intuitive, ensures that the testing and the codebase remain in sync, and automates the process of performing the symbolic tests in the same way that concrete testing schemes work in modern languages.

The other important piece of software to talk about is Ghidra. Ghidra has already been covered in the chapter on P-Code, but we mention it again here because is it immediately relevant to our work to follow. A broad grasp of what Ghidra does and what P-Code is all that is required of the reader.

4.2 What to Connect

The aim of this thesis is to perform symbolic analysis on binary programs. In pursuit of that goal, we have taken the approach of building connective tissue between Ghidra and Crucible. There are several plausible paths to do so, and each has its advantages and disadvantages.

An initial direction of inquiry might be to simply disassemble a program with Ghidra, and feed the C to Crucible. This would certainly be functional, and generally fairly ergonomic, as it would allow Crucible queries to be phrased in terms of the C names for data. However this is not the method we chose. The issues that arise from this method are twofold. The first is that there are kinds of code that are hard to represent understandably in C (even if they were originally C programs). One example

could be Linux kernel code. While one can certainly disassemble the resulting binaries (most of the time), there are many tricks in the original source that may be obfuscated by the disassembly process. An obfuscated but correct representation is often worse than an inelegant but direct representation in these cases. The second is that Ghidra is a extremely large and complicated program. We deemed it better to write a more isolated and direct program, as the use case for symbolic analysis on binaries is not completely overlapping with the reverse-engineering aims of Ghidra.

Instead, we chose to rely minimally on Ghidra. After Ghidra produces P-Code for the binary, it is no longer required at all. We deem the conversation to P-Code a light dependency on Ghidra, as it would be possible to implement one's own ISA to P-Code translation program for all of the ISAs under consideration if one didn't want to rely on Ghidra at all. The justification for using P-Code is that it is extremely general, at it allows our program to be agnostic of initial ISA.

On the Crucible end, the natural access point is to write a P-Code back-end for Crucible, which is exactly what we did. Thus upon exiting the back-end, Crucible can act on its own representation as if it were a program sourced from any other back-end.

4.3 The Artifact

Finally we have the shape of the primary code artifact of this thesis: a back-end of the Crucible library to interpret P-Code input. Specifically a Haskell program that depends on the core Crucible library and aligns with the P-Code dumping script discussed above (2.3), providing an interface into Crucible for ingesting P-Code.

At first glance, this appears to a fairly straightforward task: simply gluing two pieces of extant software together. However there are several challenges that arise from the fact that the P-Code that is fed as input only vaguely resembles the other back-ends' respective input languages. Critically P-Code lacks all of the explicit structure of the other IRs that Crucible accepts. As should be clear from the chapter on symbolic analysis, the structure of the computation being analyzed is critical to combining smaller units into larger, more useful symbolic statements.

Not only does P-Code not contain that information, even where the boundaries between units of computation are is left unannotated. Other important bits of information P-Code lacks include: the size and structure of any composite pieces of data, the number of function arguments, and all information about types. These are reasonable exclusions for an ISA, but the inputs that Crucible accepts via other backends already are IRs, and not ISAs. The differences between the two will be detailed over the course of this chapter. In short, since P-Code is a pseudo-ISA, it lacks the isolation and abstraction of a something resembling a functional programming language, a lambda calculus, or an abstract grammar, but it also lacks the explicit structural information that a compiler IR like LLVM or Mir, such as control flow annotation. Crucible's internal language is a combination of both, but leans heavily towards the former. As such, to interpret P-Code it must be presented as something similar to a compiler IR. However, as stated, P-Code lacks this information. The solution to this

conundrum is broken into several parts:

4.3.1 Function Arguments, Data Shape, and Types

The related problems of the arrangement, size, and shape of composite data structures, and the types of individual variables are completely absent in P-Code. While conjectures can be made in some cases by examining the usage, this problem is too difficult to solve in general. For instance, sophisticated programs can often use data of a single type in multiple ways. An example could be a sequence of bits stored in source code as a 64 bit integer. However depending on the interpretation of that data in the program, it may very well be accessed as a 64 bit integer, two 32 bit integers, or any number of other arrangements. Critically, a pair of completely unrelated 32 bit integers that are both acted on independently could realistically be placed next to each other. Here we can see that even in this toy example, the decision about how to talk about this stretch of memory is complex.

These difficulties are further exacerbated by the fact that P-Code does not distinguish between type. The same stretch of data only has a type (is interpreted according to some encoding) for the duration of an instruction. This means that a location can be manipulated as both a floating point number and an integer without explicit conversion. In many languages this is not a legal construction, despite the fact that it can lead to useful computation (the fast inverse square root being the canonical example W. Kahan [1986]). While this is not uncommon in binaries and is thus representable in various IRs, there one would likely have additional type information, explicit type casts, or other auxiliary information.

These facts demand a change of mindset, and luckily one that aligns cleanly with other aspects of the setting. Instead of trying to produce types and structure for data, we take P-Code's approach and leave everything untyped. While this can create difficulties in terms of referring to data of interest, we again take the stance that it is better to be direct and inelegant than to be streamlined and potentially incorrect or misleading. This mindset is taken to varying degrees by all the tools that work directly with program binaries. In general the standard is to do best effort typing, but to leave things untyped when ambiguous. For instance, a register that only participates in integer arithmetic instructions of a uniform width can be typed fairly safely, but one with mixed widths or encodings cannot.

This change of mindset extends to function arguments, though the situation is slightly more straightforward. In general, functions in a program conform to some specific **calling convention**. A calling convention is a set of shared responsibilities between the caller and the callee of a subroutine. They determine questions like where to expect input arguments, where to place returned outputs, which locations are safe to modify, and which should be left alone. While not included, a reasonable guess could be made about the calling convention for a function in P-Code. However we intentionally choose not to make such an assumption. Instead, we take the approach described in the symbolic analysis chapter 3.1.4, and treat the function as taking the entire register set as an input, and producing a new one. This captures side-effects in registers that are delegated scratchpads for callees, and other pieces of information

that are not regulated by the calling convention. While rare, these extra-conventional effects are sometimes used productively. This formulation of functions captures those changes that would otherwise be missed. In addition, the cost is small, as it is clear to see that if a register doesn't change between the input and output register sets of a function, then the function must produce no effect there.

4.3.2 Control Flow

The challenges of using P-Code as a computational language discussed in the previous section are all fundamentally questions about the representation of data. While good solutions to these questions are important, fundamentally one can always fall back on a fundamental representation: a sequence of bytes. However when it comes to control flow, there is no such nicety.

The control flow of the target program's execution must be represented explicitly. However, as stated, P-Code completely lacks these representations. Instead, it contains exactly the information that an ISA normally would for control flow instructions: the target of the jump, the register to treat as the target for indirect jumps, and possibly other registers for conditional jumps.

Note that this information is enough to execute the program, and thus for any singular execution of the program on concrete inputs, it completely defines a sequence of control flow actions. However Crucible expects a representation of control flow that captures all possible paths.

Upon receiving P-Code, our translation layer must provide to Crucible structural information that it does not possess and did not take as an input. Thus the first major action that our translation layer must perform is the reconstruction of that structure so that it can be passed to Crucible.

More rigorously, Crucible expects a Control Flow Graph. A **Control Flow Graph**, or CFG, is a directed graph in which each vertex or node is a computation that is atomic (indivisible) with respect to control flow, and each edge is a possible transition between vertices, annotated with whatever extra information is appropriate depending on the nature of the transition. A subset of the instructions of a program are said to be atomic with respect to control flow if they are sequentially contiguous, and have the following properties under any execution of the program:

- The instructions must be executed in program order.
- If a single instruction is executed, they must all be executed.
- A control flow instruction (a jump or branch) can only appear as the final or "terminating" instruction, if at all.

Note that this method of characterizing control flow atomic computations is not the minimal definition, and in fact contains non-trivial overlap, but is meant to give the reader intuition about the core idea. That being that one such interval of instructions or **block** as they are often called, can only be entered at the top, and can only be exited at the bottom after passing through all the instructions between. With that

property in mind, it follows to build a directed graph out of them, as they have clear entries and exits.

```
1   matrixMultiply (a,b,
        c,N) {
2     for (i in [0,N)) {
3       for (j in [0,N))
            {
4         tmp = 0;
5         for (k in [0,
              N)) {
6           tmp = tmp +
7           ((a[i][k]) *
8           (b[k][j]));
9         }
10        c[i][j] = tmp;
11      }
12    }
13  }
```

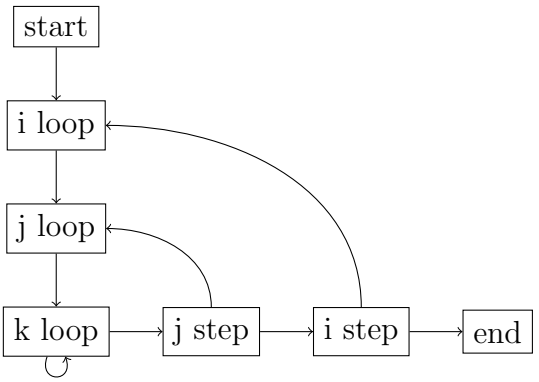


Figure 4.1: NxN Matrix Multiply and Corresponding CFG

From this definition one can see that any concrete execution of a program is some walk along the edges of the CFG of the program. Execution starts at some distinguished start node, and proceeds through some number of atomic nodes, following the transitions from one to the next before finally reaching some node with an exit condition. Often the final nodes are collected as a single distinguished end node to mirror the initial start node. Note that not all walks on a CFG are possible execution paths of the computation. This is because the CFG captures only the flow of the program's execution, and optionally some meta-data about the conditions for some transitions. It does not contain any information about the values of inputs or outputs of intermediate computations. The CFG may contain a node with a loop transition to itself with the condition that `x1` $\neq 0$, but without knowing what the computation of that block is, it's impossible to know in general what number of loops a legal computation can perform.

4.3.3 Creating Crucible CFGs

Converting our own internal representation of CFGs to Crucible's representation is the final step in feeding P-Code into Crucible. This process is made non-trivial by a few differences between Crucible's internal SSA-like computation language, and P-Code's Pseudo-ISA nature.

The first, most immediate issue is that P-Code makes heavy use of location reuse, i.e. registers are not single-write. There are canonical conversions from forms like P-Code to SSA forms (recall static single assignment 3.1.1), but luckily we do not need to implement them ourselves, as Crucible provides a "Registerized" CFG

format, that allows describing computation in terms of mutable registers, and can be converted to the standard "Core" CFG form internally. Even with this concession, P-Code's conception of registers and Crucible's are not really aligned. Specifically, P-Code does not describe individual registers, instead having register operations act on intervals of a `register` address space. Because of this, register operations in P-Code in full generality have all the issues of classical memory operations in static analysis, which we will discuss later. To make the problem feasible, we make some common but simplifying instructions. We instead operate under these (checked) assumptions:

- Each register is 64 bits wide.
- A single varnode contains data from at most one register.
- Registers are disjoint; writing to any register will not change the contents of any other register.
- Registers are indexed. Register i contains the 8 bytes starting at varnode `register` offset $i \times 8$ (in bytes).

It should be clear that this provides a *register, offset* pair for each byte of the `register` address space. Furthermore, that these pairs are all distinct. Thus this represents an unambiguous naming scheme for the data locations there.

These assumptions together with the naming scheme given above let me describe P-Code's generalized interval-based register operations in terms of concrete registers in the way that most ISAs operate. Critically, this is also how Crucible represents registers: disjoint mutable locations that can accessed directly from a handle (index). A moment should spent to discuss the second assumption, namely that access are confined to a single register. This assumption exists to restrict the access pattern to something that matches Crucible's pattern. However, this also effectively eliminates the generality that P-Code creates in the interval representation. This is still a reasonable restriction to place as very few real ISAs have operations that would be encoded in this manner. An example could be an `x86` operation that performs multiplication and places the high and low words of result in adjacent registers. While not inconceivable, this is not a common practice and not core to most ISAs.

If these assumptions become too limiting in the future they can be relaxed. The single register access assumption can also be lifted, though it requires some sophisticated handling of registers. In full generality, with unaligned and multi-register access, this is a serious increase in complexity that may also introduce other limiting factors, such as requiring endianness knowledge. Other solutions could include mixed-width Crucible registers. Given the difficulty of full generality, and its relative infrequency of use, we chose the assumptions above as a middle ground of expressiblity and simplicity.

Luckily, P-Code's conception of operations not being bound to specific data widths lines up nicely with Crucible's flexible width operations. Some conversions are required, such as sign or zero extending certain inputs, but in general both languages take the position that operations like addition and bitvector concatenation are defined for all sets of input and output widths that make the operations unambiguous and

information persevering. (e.g. one can't concatenate bitvectors with lengths a and b into a result location narrower than $a + b$.) Some massaging of data is required, but generally both systems store the width information of an operation in the operands rather than in the operation itself, which is different than traditional ISAs.

With all of this in mind, we now have enough structure to translate individual P-Code pure register operations. For an instruction that takes only registers as inputs and registers as output, we can now translate those register intervals to slices of Crucible registers (checking our assumptions along the way), and feed Crucible (symbolic) values obtained from the registers into the appropriate operation, placing the result back in the register indicated by the instruction. In reality there is a lot of data shuffling behind the scenes to take only a slice of a register as input, or write back only a subsection of a register without clobbering the rest of the register. In addition, there are non-trivial issues surrounding Crucible's notion of what appropriate inputs (mostly input widths) are for operations. These restrictions on appropriate inputs require Haskell's dependent types, which will be discussed in a later chapter 6.3.

Finally, the linking of nodes in the CFG is largely free, as during the construction of our internal CFG representation, we already check and enforce the restrictions that Crucible expects, such as only allowing control flow instructions in the terminating position. Thus the overall graph form of our internal CFG and our desired Crucible CFG are the same. There are several small issues remaining, namely Crucible's Block Argument passing style, and the issue of Memory, which will be discussed in the next two subsections.

Putting aside those two issues for a moment, what has been described so far is enough to capture register only programs that only use indirect jumps for function calls and returns. This is already captures a large number of programs, but the average small C program is still not fully represented, as the stack is in memory. The next three subsections detail some of the challenges presented in the handling of such programs before the final subsection presents the P-Code backend in its final state.

4.3.4 Reconstructing CFGs

Armed now with a grasp of CFGs, we can now proceed to building a CFG to represent a given program in P-Code. There are several rules that immediately suggest themselves for dividing a stream of instructions into blocks. These are:

- The first instruction of a function or subroutine is the head of a block.
- If an instruction is a control flow instruction, then the immediately following instruction must be the head of a block.
- If an instruction is pointed to by some control flow instruction, then it must be the head of a block.

These simple rules do most of the heavily lifting of separating an instruction stream into a sequence of blocks. However this is also an occasion where we need to limit the scope of our tool. The problem comes with the combination of indirect jumps with

the rules given above. An indirect jump, as described in the Prerequisites chapter 1.1.5, takes the value of a register and execution of the next instruction begins at the location indicated by the register. Furthermore, knowing where a register might point is not a trivial problem, even in fully deterministic programs. Fortunately, in reality indirect jumps see fairly limited use in C-like languages. Modern languages use them more, but C itself, which is the primary target of interest, uses them in very limited ways. In C they appear in explicitly higher order functions (functions in which functions are arguments), and sometimes in switch statements, which direct control flow based on a single value and allow more than two subsequent blocks. In modern languages, constructions like virtual function tables or dynamically typed objects call for indirect jumps generally. While such paradigms are expressible in C, they are not a core language feature and thus appear far less frequently.

The primary way that indirect jumps are used is in returning from subroutine calls. Fortunately this usage is much easier to model than general usage. We will return to the question of subroutines and functions momentarily. In our construction of CFGs, we handle only the case where all indirect jumps are for function and subroutine purposes. This assumption is checked explicitly and is used only for ensuring that the set of edges in the final CFG is complete. This means that should this assumption prove too limiting in the future, the CFG creation process can be augmented to perform *points-to analysis* or other static analysis techniques to loosen this narrowed scope.

In the narrowed scope of only direct jumps, optionally conditional, continuing to put aside subroutine returns for the moment, the rules above give a complete set of blocks for a program. Now to connect them:

- A block with a non-control flow terminating instruction has a single successor block, namely the block next in program order. This block is headed by the instruction immediately following the terminating instruction of the current block.

- A block with a non-conditional jump as the terminating instruction has a single successor block, headed by the instruction indicated by the target of the jump.

- A block with a conditional jump as the terminating instruction has two possible successor blocks: the following block in program order, and the block indicated by the target, as above.

With these rules, we now have a mostly complete CFG creation algorithm. We give an example here of a RISC-V function that tests if an integer is prime by attempting to find factors one at a time, along with its corresponding CFG. The arrows without destinations on the right represent subroutine returns.

Listing 4.1: Primality testing in RISC-V

```
# Our number to test is in register a0, and the caller expects the
# result, 1 if it is prime, 0 otherwise, in a0. We will want to call
# other things, so we need to save a few numbers out of the way. We
# make space for them on the stack, a place in memory for storing
# temporary values.
isPrime:
addi sp, sp, -16     # Make space for 4 4 byte values
li t0, 2             # Initially a = 2
outer_loop:
mv t1, t0

inner_loop:
st ra, (sp)
st t0, 4(sp)
st t1, 8(sp)
st a0, 12(sp)        # Registers that might get clobbered are safely
                     # saved

mv a0, t0
mv a1, t1
call multiply
# This is another function we are calling out to. It takes two numbers
# in a0 and a1 and returns their product in a0. It's definition has
# been elided for brevity. In most real computers, there would be an
# instruction for this, but the most minimal RISC-V set does not have
# one.

mv t2, a0            # This is our product to test
ld ra, (sp)
ld t0, 4(sp)
ld t1, 8(sp)
ld a0, 12(sp)        # Restore our saved values

beq a0, t2, found    # Our product matches
bgt t3, a0, too_big # Our product is too large
addi t1, t1, 1       # Need to keep going, increment b and try again
j inner_loop

found:               # We found a counter example
mv a1, x0            # Not prime
addi sp, sp, 12      # De-allocate the space we allocated
ret

too_big:             # Our product was too big, are there more to test?
beq t0, a0, done     # a = n, we are done
addi t0, t0, 1
j outer_loop

done:                # Failed to find any counterexamples
mv 1, a0             # Is prime
addi sp, 12, sp      # De-allocate the space we allocated
ret
```

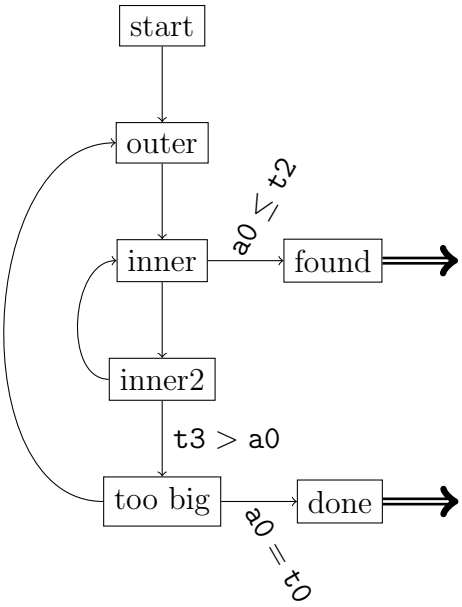


Figure 4.2: Primality Testing Corresponding CFG

Now to the question of subroutine returns. This is again a question of narrowed scope. Here our assumption is that any control flow instruction that is not a subroutine call or a subroutine return is limited to targets within the current function. This is a very reasonable assumption, and easily checked. This assumption is expected directly by Crucible, so there is no question of leaving space for the future, as there was with indirect jumps. However this assumption covers the vast majority of C-like programs, with the only exceptions being particularly esoteric ones.

Armed with that assumption, then the CFG construction rules presented above together with a drop-in marker for subroutine calls and returns, are enough to create a CFG for any single function.

Crucible's IR has the form of a directed graph with two levels of hierarchy. The outer graph is a function call graph. Each vertex is a function, and it has an edge to any function that it calls. (Note that this is not enough to assume any nice structure such as being acyclic.) Each vertex of the outer graph also has a CFG associated with it, the inner level of the digraph, which represents the control flow internal to that function. That internal CFG has subroutine call instructions that represent the out-edges of the associated vertex, and generally at least one subroutine return that is represented by reversing whatever in-edge of the associated vertex that brought the current execution to the current vertex.

Finally we have a very simple example of a program with more than one function and a two level CFG. Most programs would have far more complicated relationships between their functions (including self loops or cycles) but this example is a simple three function tree.

Listing 4.2: Pseudo-code For Orthogonal Matrix Conjugation

```
1  # multiply a and b and place results in c
2  matrixMultiply (a,b,c,N) {
3    for (i in [0,N)) {
4      for (j in [0,N)) {
5        tmp = 0;
6        for (k in [0, N)) {
7          tmp = tmp +
8          ((a[i][k]) *
9          (b[k][j]));
10       }
11       c[i][j] = tmp;
12     }
13   }
14 }
15
16 # conjugate a and place it in b
17 transpose(a,b,N) {
18   for (i in [0,N)) {
19     for (j in [0,N)) {
20       b[i][j] = a[i][j];
21     }
22   }
23 }
24
25 # place a'ba in c, using t as a temporary
26 # where a' is the transpose of a
27 orthogonalConjugate(a,b,c,t,N) {
28   transpose(a,c,N);
29   multiplyMatrix(c,b,t,N);
30   multiplyMatrix(t,a,c,N);
31 }
```

Applying all of the above together brings us to the P-Code back-end's internal conception of CFGs. Next we must feed those CFGs into Crucible.

4.3.5 Block Arguments vs SSA

A small wrinkle is the question of how data progresses between CFG blocks. This question arises in the following situation. A block C has two possible predecessor blocks, A and B. Both A and B write some value to a location `x1`, which is the used as an input to some assignment in C without an intervening assignment to `x1`. In short, C depends on the value in `x1`, but that value is produced (differently) on each path to reach C.

In a classic SSA form CFG, data within each block is manipulated and transmitted through a string of assignments, and when a block has more than one predecessor, a **phi-node** is placed before any assignment that uses a contested location like `x1` above. Recall that in SSA forms, a single data location is split into versions. So the question is not the value of `x1`, but which version of `x1` to use? Specifically, should the assignment in C that takes `x1` conceptually use $\texttt{x1}_A$ or $\texttt{x1}_B$? A phi-

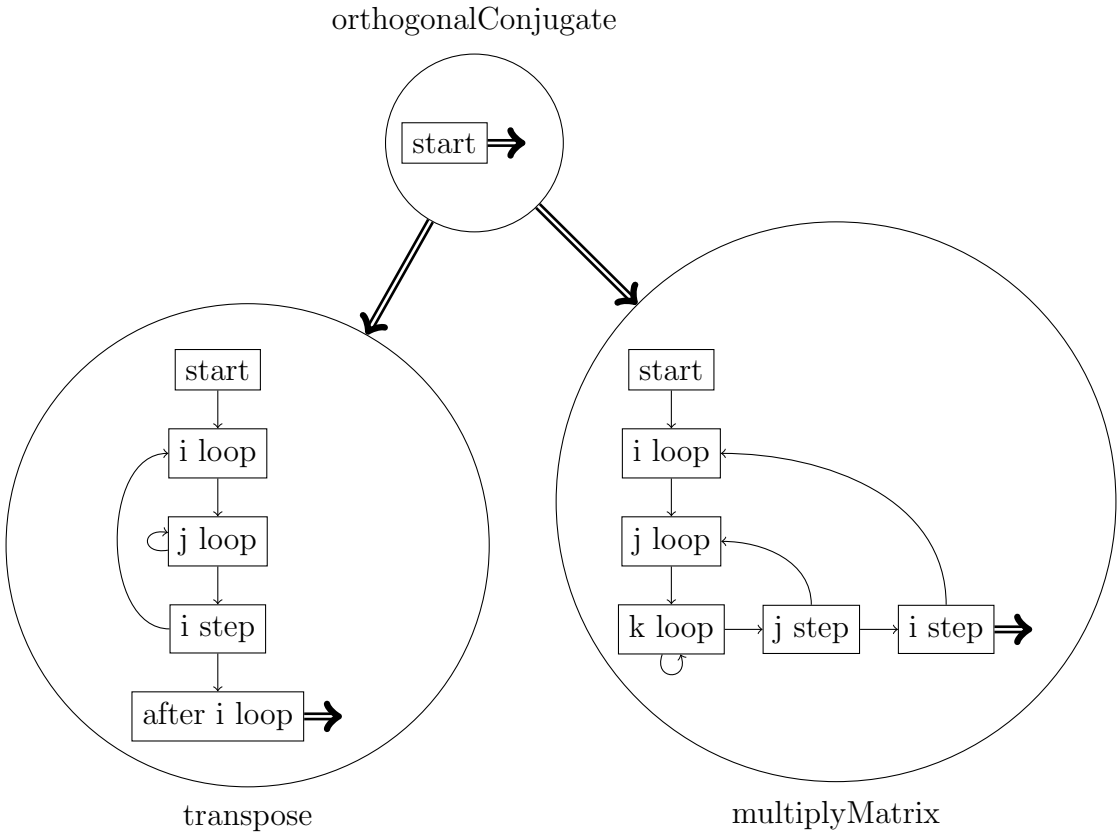


Figure 4.3: CFG for Orthogonal Matrix Conjugation

node is a special operation that takes p location versions (generally all the same conceptual location prior to version splitting), and produces as output the input associated with the predecessor block for this particular execution path through the SSA CFG. Continuing our example above, the statement

$$\texttt{x1}_{C0} \leftarrow \phi(\texttt{x1}_A, \texttt{x1}_B)$$

placed in block C means "Let the initial value of `x1` in C be $\texttt{x1}_A$ if the last block was A, and $\texttt{x1}_B$ if the last block was B."

This captures a fairly natural intuition of anyone who has programmed in any imperative language. If a variable does not go out of scope when exiting an optional or conditional block, the value after said block in the surrounding scope depends on whether the block was executed. It's also worth noting that despite this exact principle being in effect, P-Code, like all assembly languages and ISAs, does not explicitly encode this information. This is partly because one reason to use phi-nodes is that they make data dependencies in the CFG explicit. In P-Code, all blocks can view and act on the entire machine state (i.e. every block can touch every register), so dependency tracking is not the focus. Even if a block doesn't touch a register, it could, and so this sort of dependency tracing is not present in Low P-Code.

Crucible makes use of a different system to accomplish the same goal of dependency tracing. It instead uses block argument style. **Block Argument Style** replaces phi-nodes by attaching a set of arguments to each register. A block has an argument for each SSA location version that is contested. So there is exactly one block argument for each phi-node one would include in a traditional SSA. Then a block performs what is essentially a tail call of the next block, instead of a jump as in a traditional SSA. Returning to our example above, assuming `x1` is the only contested location between these blocks, then C has a single argument $x1$. Where A had `JUMP` C in a traditional SSA, it now has `CALL` $C(\texttt{x1}_A)$, a call with a single argument corresponding to the value the argument should take in the body of C if it was reached through A. Inside C, statements can use the arguments as inputs to assignments like any other SSA term. This example is an abuse of notation, but meant to give the intuition that block argument style makes data dependency explicit at both the caller site and the definition of destination block. Instead of collecting all the phi-nodes in the block, instead all contested dependencies on prior blocks are explicit in the arguments.

Again, P-Code lacks all explicit dependency information, whether phi-nodes or block arguments. So either way we would need to reconstruct this information. Reconstructing block arguments given a CFG is not very hard. Simply walk the graph, and for a node, every location that it observes without first writing to is a dependency. A block has as arguments all of its own dependencies, and the union of the dependencies of all blocks that are reachable through the block in question.

However for convenience, we will be emulating P-Code's uniform approach. Specifically, instead of tracking block dependencies, every block takes the full register state as arguments. Thus the block arguments for P-Code translated this way are overly conservative, as many registers will be included as dependencies that are not in fact in use. This same approach will be extended to functions as well. This follows our

general approach to ensure the inclusion of all effects of a computation rather than using a more streamlined or tighter fitting approach that might miss some effects.

4.3.6 Memory Issues

For our purposes the differences between memory and registers are twofold. The total number of data locations between all the registers is much much smaller than that of memory, and that memory permits indirect access. What that means is that for an operation on registers, the set of involved registers is known exactly by simply examining the program. Memory on the other other hand, is addressed indirectly. This means that simply looking at the program alone is not enough to determine where memory is affected. The classic load and store instructions in many ISAs, and which P-Code models, take a register as an input that determines the location in memory that data is being moved to or from. We say that the register indexes memory.

This greatly complicates analysis. With registers, if an operation takes a register as input, one can simply look up what is known about that register's contents. On a load from memory into a register however, one can't simply look up what is known about that location, since which location to look up is also a term in the computation. If one gets lucky and during analysis we know that the indexing register can only contain a single value, then the lookup is still possible. Consider the case where the indexing register could be one of two values, i or j. Then the most precise statement about the contents of the register that was written during the load is that it contains `Mem(i)` if the indexing register was i, and `Mem(j)` if it was j. Those lookups likely contain complex symbolic restrictions, and much of the time, the indexing register will be far less constrained. It's clear that this will lead to a blowup of symbolic complexity even on simple programs. Even simple operations, such as writing to a location, and then reading back that very same register may be complicated to resolve, despite the fact that it is clear that they should result in the same constraints.

There are several approaches to this problem. While difficult, it is not completely intractable, and there is a developed field, **points-to analysis**, that tries to answer this class of questions. One can find an account of basic points-to analysis in Schwartzback. One approach in other parts of Crucible is to postpone these problems until symbolic execution time, and keep a list of memory locations that have been written to. On a read, walk that list to find a matching write for each possible value under the constraints on the indexing value, and return that set of values. This is correct, but extremely expensive in many cases, particularly for programs with many memory accesses and non-local access patterns.

The method we will be using is simpler but more crude. Any write to memory succeeds automatically. Any read from memory produces a completely unconstrained symbolic term. In short, in our analysis, every memory access could produce any value, even in trivial cases like a write followed by a read to the same location. This is much faster, but obviously produces extremely over-conservative symbolic constraints, as large amounts of information are being destroyed. Due to the overly conservative nature of this analysis, it becomes impossible to establish useful bounds on many

programs, particularly those with relatively few subroutines and large working sets. Regardless, it remains a useful strategy for many programs of interest, particularly when combined with a human who can look at the binary and manually provide assumptions.

4.3.7 Final Artifact

The artifacts of this thesis are twofold. The first is plug-in for Ghidra that dumps the P-Code of each function into a file. This script was discussed in the chapter on P-Code and was a relatively straightforward change to some existing plug-ins.

The more interesting of the pair is the backend for Crucible. With the concepts in this chapter, we can now express completely what the backend does. Specifically, it is a Haskell program that relies in the Crucible core library and performs translation of P-Code into a Crucible CFG, suitable for making symbolic queries on. This process is composed of a number of steps. First is parsing the P-Code from Ghidra. This simply entails reading in the file containing the P-Code and transforming it into a list of functions, each composed of a list of instructions with associated addresses and operands. Next is reconstructing the CFG for each function. We do this with the same algorithm discussed in the symbolic analysis chapter 4.3.4. This produces a directed graph for each function. We perform further checks and manipulations to produce the suitable CFG. For instance, we check that no block is terminated with an indirect jump (excluding returns), allowing us to be assured that we have the full list of CFG blocks and we are not accidentally treating two blocks as connected. After finalizing our CFG, we annotate edges with the appropriate information (conditionals, etc), we implement back edges, allowing the digraph to be traversed in reverse, etc. This annotated CFG is then suitable to be fed into Crucible.

Our various simplifying assumptions from previous chapters are still in effect. We model memory reads as unconstrained symbolic values, we do not support indirect control flow, and so on. Thus our tool is suitable for operating on primarily register-bound programs, and will be of little to no use for programs with high memory usage or a large working set, forcing the use of memory as a swap space. Limited though it may be, These limitations have been implemented at strategic locations so that if the assumptions are to be lifted in the future, a full rewrite will not be necessary, a module that handles memory accesses, for instance, has a natural place to slot in.

Stated as such, however, this program seems far simpler and easier to write than it actually is. This is because while the above is a full description of what the program does, *how* these things happen is actually the source of the majority of the complexity of this thesis. This complexity is introduced by Crucible, as it is written in a particularly strict paradigm of Haskell. As learning and producing this style was a major component of the work on this thesis, the remainder of this document is dedicated to discussing some of theoretical techniques and paradigms in play in this project. We are particularly interested in **dependent types** and **monadic** style Haskell, which we will define and motivate in the next two chapters.

Chapter 5

Type Theory

> God bestows to his most patient of children only the most agonizing of tasks. Not for our inherent suffering, but for our finesse and skill unparalleled - to serve the people.
>
> Ozymandias Juarez

In order to understand the power of dependently typed Haskell, we first need to have some idea of what a dependent type is. To that end, this chapter is an overview of one formulation of several flavors of type abstraction, and what one might use them for. This chapter is written for a reader that has had some exposure functional programming and lambda calculus, but the ideas introduced should be understandable to any reader.

Specifically, this chapter is a light treatment of the hierarchy of type systems known as the Lambda Cube. All the systems of the lambda cube are based on the simply typed lambda calculus. The reader is expected to have basic familiarity with the original untyped version of the lambda calculus. Those unacquainted can find an overview here Horwitz. The basic mechanics of the creation and application of lambda abstractions and the concept of equivalence up to beta reduction and eta reduction is all that is required of the reader.

5.1 Simply Typed Lambda Calculus

The **simply typed lambda calculus** is an extension the the original untyped lambda calculus in which each term in the calculus has an associated type. Informally, the **type** of a term is a piece of meta-data that describes the shape or interpretation of the term inside. In practical programming languages these types are informative in terms of how to interpret the contained data. In these more formal languages however, types serve to separate collections of terms from one another. We will see in a later section that types in these more abstract contexts are not without interpretation either. The simply typed lambda calculus extends the base untyped lambda calculus

in one straightforward manner: each term must have exactly one type which does not change across appearances of that term, and types must match when substitution occurs.

When introducing a term, we write $x{:}t$ which is some term x with type t.

There must exist a base set of types B, each of which has some associated set of constants T_c. These form the atoms of the type. For instance a type could be nat, the type of natural numbers. Then the constant set for nat would be the set of natural numbers. Let $T \in B$ be some type. Each type in the simply typed lambda calculus must either be a base type or combination types formed via functional abstraction. The syntax of a completely general type is then:

$$\tau \stackrel{def}{=} \tau \to \tau \mid T$$

Where $a \to b$ is the type of a lambda abstraction that takes a term of type a and produces an expression of type b. Terms change compared to the untyped version as well:

$$e \stackrel{def}{=} x \mid \lambda x{:}\tau \,.\, e \mid ee \mid c$$

These are a reference to a variable, the introduction of an abstraction (now with a mandatory type annotation), the application of an abstraction, or a constant respectively. For example a construction that increments a natural number (assuming a $+$ operator defined on naturals) could be

$$inc \stackrel{def}{=} \lambda x{:}nat \,.\, (x + 1)$$

where $inc : nat \to nat$.

We will not present the typing rules in full detail, as they are symbol dense and not greatly helpful. In prose they are,

- Constants have the type they ought to.
- If term $x{:}\sigma$ would imply the existence of a term $e{:}\tau$, $(\lambda x{:}\sigma \,.\, e)$, the abstraction that takes some σ x and substitutes it with e must have type $\sigma \to \tau$. It takes a σ and produces a τ.
- If you have an abstraction $e_1{:}a \to b$ that takes some type, and a term of that type $e_2{:}a$, then the application $(e_1 e_2){:}b$ is well typed as expected.

The simply typed lambda calculus has the same reduction rules as its untyped cousin, restricted to statements with appropriate type. Namely

$$(\lambda x{:}\sigma \,.\, t)u =_\beta t[x \leftarrow u]$$

holds for $t{:}\tau$ and $u{:}\sigma$, as one would expect from the rules above. Similarly for eta reduction:

$$\lambda x{:}\sigma \,.\, tx =_\eta t$$

when $t{:}(\sigma \to \tau)$ and x is not free in t. Operationally, the choice of evaluation strategy is the same as the untyped version.

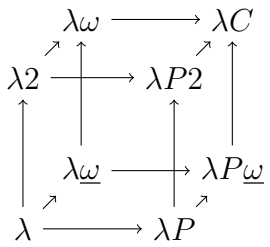


Figure 5.1: Lambda Cube

Moving away from pure formalism, the reader might note that the only kind of abstraction in this version is $\rightarrow$. This makes for somewhat clumsy computation, as it doesn't capture the notion of associating terms into compound terms unless said compound term is also a base type. Additionally, there is no way to define an abstraction on more than one type at a time. These are among the concerns that systems extended from simply typed lambda calculus aim to address.

5.2 Lambda Cube

This account of the lambda cube is drawn from Barendregt [1991].

The lambda cube is a collection of eight related type systems that are naturally structured as a cube in three dimensions, with one system at each corner. The dimensions are associated with a kind of abstraction in the type level of the system, with the low end of each dimension lacking said abstraction and the high end allowing it. The three kinds of abstraction are independent, and thus all eight corners are occupied and distinct. The systems also have a natural partial order structure, with the systems on the high end of each dimension include the system on the low end of the dimension. This follows immediately from the system with the more flexible types being at least as expressive as the same language without the additional abstraction. An image of the lambda cube is presented to help understand this relationship.

The close bottom left corner labeled with λ is the simply typed lambda calculus described above. Proceeding upwards allows for the creation of terms that depend on types. Proceeding into the page allows for types depending on types. Proceeding right allows for the creation of types depending on terms.

A term depending on a term is universal for all systems of the lambda cube. It is the sole kind of abstraction present in the simply typed lambda calculus. A lambda abstraction term $x{:}a \rightarrow b$ when applied to some term $v{:}a$ produces a $(xv){:}b$. The resulting term (xv) of the application depends on the choice of input term (v).

5.2.1 Universal Qualification

Another piece of formalism we need is product terms (and later product types). We want to extend our basic syntax with the symbol Π, which we can use like so

$$\Pi x{:}A$$

which means "for all x of type A." Note that alone this is not a valid construction in any of the lambda cube systems, but we will see that it has intuitive use in the following abstractions.

For example we can re-express the "type" (informally) of functions from type A to B as

$$\Pi a{:}A \;.\; B$$

or "for all terms a of type A, some term of type B."

5.2.2 Polymorphism

An example of a term that depends on a type is a **polymorphic function**. Polymorphic functions are functions that when applied to a type produce different term-level functions. A natural example could be the identity. Every type a has an identity function $I_a{:}a \to a$ that takes each term with type a to itself. However it's natural to think of a single unified identity I. This polymorphic version can be applied to any type t and produces I_t, the identity on that type. This abstraction is notated formally with a Π universally qualifying over a type like so

$$I \stackrel{def}{=} \Pi e{:}* \;.\; \lambda x{:}e \;.\; x$$

which says that for all types e, I takes a term of type e to itself. Here $*$ represented the set of all types. This is discussed in greater depth in the chapter on Haskell. This qualification is sometimes written with a Λ instead of a Π for specifically universal qualification over types, but we will avoid this notation to prevent an explosion of new symbols. Often when using Λ, the type will be supplied explicitly as an argument. Again we will avoid this explicit notation to simplify expressions. No ambiguity is introduced, as the term arguments are of concrete type and thus the type specialization can be inferred.

5.2.3 Type Constructors

A type that depends on a type is called a **type constructor**. This follows the intuition that it is a function that takes a type and produces a type. Thus one can construct a new type out of existing types. At the simplest level, this abstraction allows one to write $A \to B$ where A and B are types. A motivating example could be pairs where the two elements contained must all be of the same type. Consider the following encoding (from Horwitz) of a pair type using an access function where

$\mathbb{B}$ is the type of booleans, defined like so

$$\text{TRUE}{:}\mathbb{B} \stackrel{def}{=} \Pi e{:}* \,.\, \lambda x{:}e \,.\, \lambda y{:}e \,.\, x$$
$$\text{FALSE}{:}\mathbb{B} \stackrel{def}{=} \Pi e{:}* \,.\, \lambda x{:}e \,.\, \lambda y{:}e \,.\, y$$

We can see that both true and false are polymorphic functions on two elements. True returns the first argument, ignoring the second, and false does the reverse. With these definitions we can define

$$\text{pair} \stackrel{\text{def}}{=} \lambda \text{e}{:}* \,.\, \lambda \text{i}{:}\text{e} \,.\, \lambda \text{j}{:}\text{e} \,.\, (\lambda \text{s}{:}\mathbb{B} \,.\, \text{sij})$$

The explicit argument e is some type, and the resulting term is a function that that takes the two terms of the given type, and returns a function that returns either the first or the second depending on the given boolean. This formulation requires polymorphic functions, but one can use type constructors without them. The two kinds of abstraction are often found together but not fundamentally interdependent. In the standard style, the right side of this definition would be

$$\Lambda e{:}* \,.\, \lambda i{:}e \,.\, \lambda j{:}e \,.\, (\lambda s{:}\mathbb{B} \,.\, sij)$$

Which indicates that e is a type and not a term.

5.2.4 Dependent Types

A dependent type is a type that depends on some term. For a motivating example that will become relevant later, consider one formulation of restrictions on the naturals in a typed lambda calculus with both type constructors and dependent types. This definition is simplified for the reader and original, but is inspired by the formulation of similar structures in the Haskell library `parameterized-utils`, Inc. [2024] which is closely related to Crucible.

Consider a union type that we will call Either. A union type U is a type formed by some collection of other types $t_1, t_2, \ldots$, where each term $x{:}U$ must be equivalent to some term $v{:}t_i$ for some t_i in U. Our definition will restrict to two internal types. Canonically these types are associated with left and right. Consider the type constructor for a union between types l and r

$$\text{EITHER} : (\text{l}{:}* \to \text{r}{:}* \to \text{E}_{\text{lr}})$$

with

$$\text{LEFT} \stackrel{def}{=} \Pi l{:}* \,.\, \Pi r{:}* \,.\, \lambda v{:}l \,.\, e{:}(\text{EITHER lr})$$
$$\text{RIGHT} \stackrel{def}{=} \Pi l{:}* \,.\, \Pi r{:}* \,.\, \lambda v{:}r \,.\, e{:}(\text{EITHER lr})$$

as functions that create an E_{lr} term given a l or r term respectively.

This definition of Either uses type constructors and universal qualification, but not dependent types. Note that a function that takes an E_{lr} term as an argument must be defined for both a left and a right inhabiting value, though often one of those definitions may be the identity function (canonically the left type).

Consider an attempt to define subtraction on the natural numbers. Subtraction is a naturally useful operation, but one runs into difficulties due to the natural numbers only proceeding infinitely in the positive direction. What should the value of $2 - 3$ be? One can use constructions like Either to define an operation that can fail or more sophisticated embeddings of the naturals (following Peano arithmetic for example), but with dependent types one can create a natural constraint at the type level to encode this restriction around subtraction.

Subtraction is defined on the natural numbers if the second argument is at most as big as the first. So a natural tool would be to define a narrowed type for naturals larger than some given natural, $\mathbb{N}_{\geq n}$. With dependent types we can do just that

$$\text{NARROW} : \mathbb{N} \to \mathbb{N} \to \text{E}_{()\mathbb{N}_{\geq \text{n}}}$$

The definition is elided for brevity, but in prose this function takes a constraint n followed by some value v, and if $v \geq n$ then returns RIGHT v, otherwise a unit type (denoted ()) as a left value to represent a failure. Note that the right type produced here is inhabited all naturals greater or equal to n, and only those naturals. Thus for a value to inhabit $\mathbb{N}_{\geq n}$ is a type level guarantee that the value has said property.

From here we can define subtraction like so

$$\text{SUB} \stackrel{\text{def}}{=} \lambda\text{n:}\mathbb{N} \,.\, \lambda\text{v:}\mathbb{N}_{\geq \text{n}} \,.\, \text{n} - \text{v}$$

again simplified by the assumption that the basic type $\mathbb{N}$ has some binary operator $-$ that only has defined behavior for appropriate inputs. Note that even if the base $-$ operation produces undefined and undesirable behavior for unsuitable inputs, this dependently typed version is safe in all cases and does not require explicit juggling of Either terms. Compare this definition to another natural one that simply takes two naturals and returns an Either that accounts for the possibility of undefined behavior by returning a left unit value for unsuitable inputs.

For this simple case both are feasible definitions, but when part of a larger, more complicated system, it is often useful to move that sort of checking for valid inputs to the type level, rather than performing safety checks at every step.

Note that the narrowed type $\mathbb{N}_{\geq n}$ is just that, a type, but depends explicitly on a term n. This is a canonical use of dependent types, as order constraints like $\geq$ are common and unwieldy to express otherwise.

5.2.5 Systems of the Lambda Cube

We will not go into much detail about the exact systems of the lambda cube, as it is not really the focus of this chapter. Its explanation here is to provide notation and a formulation of some canonical patterns of abstraction. The diagram from the beginning of the chapter is duplicated here for convenience.

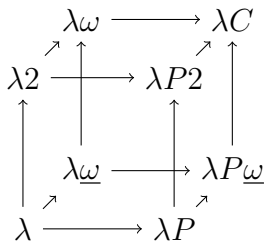


Figure 5.2: Lambda Cube

Briefly, the system in the bottom left corner, λ or sometimes $\lambda_{\rightarrow}$ is the simply typed lambda calculus, which contains none of these new types of abstraction. Moving upward, the system $\lambda 2$ extends λ with polymorphic types (terms depending on type). Into the page, the system $\lambda\underline{\omega}$ extends λ with type constructors (types depending on types). In turn, $\lambda\omega$ is more common and contains polymorphism in addition to type constructors. Similarly, the bottom right system λP extends with dependent types (types depending on terms), $\lambda P2$ with both dependent types and polymorphism. The system $\lambda P\underline{\omega}$ is quite rare and contains dependent types and type constructors, but not polymorphism. Finally in the top right, λC, or the Calculus of Constructions contains all three additional kinds of abstraction compared to λ. It largely eliminates the distinction between terms and types, and is extremely expressive.

A particular note should be made of $\lambda\omega$, as Haskell is generally considered to be equivalent with this system. As we will see in the chapter on Haskell however, it can be arranged to emulate dependent types, making it closer to λC.

5.3 Types and Proofs

Taking a step back from pure formalism, it is worth returning to the semantic interpretation of types. As discussed above, a type at its simplest is a way to provide context or an interpretation to a value or term. Types provide "shape" to terms, loosely. This is desirable both as it allows one to more easily associate meaningful semantics to the pure formal manipulations of the family of lambda calculus systems, and as it can provide helpful guard rails when attempting to formulate expressions in those systems. As types become more flexible in the ways that are described above, its natural to want to express more subtle semantic meanings to types and their relations. Arguably the zenith of that association of semantics is the Curry-Howard isomorphism Howard [1980].

The full strength of the Curry-Howard isomorphism is far beyond the scope of this thesis, however the core idea is relevant and very much in play in the code artifact that accompanies this document. Informally, it says that types are propositions, and terms are proofs. A type is said to be **inhabited** if there is exists a term of that type. An inhabited type is equivalent to a proposition for which there is a proof. To see this isomorphism more clearly, consider a sequence of functions between a variety of types. Each function is a legal manipulation of axioms or derived theorems to

transform one proposition into another. In the same way that formally a value of some type is equivalent to the sequence of functions that produced it, the final proof is equivalent to the sequence of manipulations between individual steps. Consider a type LeqProof parameterized by two naturals. An inhabiting value of that type represents a proof that the first parameter is at most as large as the second. Then a function one can define is taking a LeqProof n m along with some r and producing a LeqProof $n + r$ $m + r$. We see that with careful initial assignment of semantic meanings, the equivalence between the function that manipulates LeqProofs and the theorem that adding the same natural to both sides of a $\leq$ relation preserves the relation is immediate.

The Curry-Howard isomorphism is a bridge between standard mathematics and theoretical computer science. This is clearly visible when applying it to the lambda cube. The systems of the lambda cube are equivalent to several known systems of logic outside computer science under the isomorphism. Notably $\lambda_{\rightarrow}$ is equivalent to Propositional Calculus, λP is equivalent to Predicate Logic, and λC is (roughly) equivalent to Intuitionistic Logic, the logic of constructive mathematical proofs.

With the mindset of types as propositions and values as witnesses to the provability of the proposition, we are now armed to talk about dependently typed Haskell as it is used in the Crucible library.

Chapter 6

Haskell

> A monad is a monoid in the category of endofunctors, what's the problem?
>
> James Iry (Saunders Mac Lane)

Named after the influential mathematician and logician Haskell Curry, the program language, `Haskell`, that Crucible is written in is worth some discussion. Invented in the 1990s and continuously developed to this day, Haskell is not a particularly common language. Its use is concentrated in academia and certain corners of industry. The language stands out for being rich with meta-programming features, being extremely flexible, and serving as the site of many cutting edge ideas in programming language design.

The aspects that make Haskell stand out from many programming languages are the same reasons that it appeals to corners of academia and industry. Primarily, those are being purely functional, lazy, having rich types, and having type inference. These features do not make Haskell unique, but they, particularly in conjunction with one another, were pioneered by Haskell.

The primary aspects of interest in this chapter are the rich types and implications of a purely functional language. As such this chapter will give an account of types in Haskell while introducing aspects of Haskell's syntax, followed by the other relevant aspects mentioned above. The chapter concludes with a description and motivation of dependent types in Haskell, and how they are used in the context of Crucible.

Types in Haskell are relevant to this thesis because Crucible uses extremely rich types, even by Haskell standards. We saw in the last chapter that rich types are extremely expressive and that they are in correspondence with semantically meaningful propositions. By mixing types representing propositions and standard programming types, Crucible code is safe from a large variety of possible bugs by encoding assumptions as types.

6.1 Types in Haskell

Armed as we now are with some understanding of formal type theory, Haskell types should seem quite familiar. This section motivates types in practical programming languages as opposed to formal tools like most lambda calculus variants. Along the way we will introduce some of Haskell's syntax pertaining to types, and provide a pair of common and conceptually useful methods of understanding some key Haskell ideas.

At the most basic level, types exist in languages like Haskell to provide a context to a piece of data. Much of the time, the contents of a variable contain some information (literally as some arrangement of bits), and the type of the variable tells one how to interpret those bits, what operations suggest themselves, etc. Types exist to separate matrices, integers, and whatever other kinds of objects a program might want to manipulate. They can be thought of as a set of possible values together with a label to distinguish the values from similar ones of a different type, for instance 8 as a natural vs 8 as an integer. While they have the same semantic meaning, and are likely convertible to one another, they are in fact different, because they differ by type. The label tracks this distinction.

In discussing Haskell, "value" is preferred nomenclature to "term", but they are largely equivalent. The statements about terms in the lambda calculus chapter should be applied to values when talking about Haskell.

In Haskell, every value has a type, and can be explicitly annotated like so: `v :: t`. This means that the value `v` is of the type `t`. A function in haskell is something that takes a sequence of values of known types and produces a value of a known type. That relationship is represented as `f :: a -> b`, which says that `f` is a function that takes a value of type `a` and produces a value of type `b`. Functions can be defined on more with more than one argument. Binary operations for example are expressed as `g :: a -> b -> c`, which takes `a` and `b` and produces `c`. This process is known as **currying**, also after Haskell Curry, and can be understood either as a function on two arguments, or a function on `a` that returns a function on `b` that returns a `c`. (We say that the function operator `->` is right associative.) The idea that function types are also types is central to Haskell. So a completely generic type could be either a simple type such as `a` or a functional type `a -> b`.

Types can be separated further by kind. In the same manner as types distinguish the shape of values, **kind** distinguishes the shape of types. Every type has exactly one kind, and as with values and types, kinds do not change over the execution of a program. Kinds can be understood completely as "the type of a type". Haskell also has **sorts** above kinds with the same relationship, but currently everything in Haskell has the same unified sort.

Kinds exist to partition the space of types. In standard Haskell, there are an infinite sequence kinds, to convert the infinite collection of types in Haskell. The most basic kind is `*` or `Type`, which both refer to the collection of all zero-arity types. **Arity** refers to how many arguments something takes before it can be evaluated. A function that takes one argument has arity one, a unary function along with an argument together have arity zero, a function that takes two arguments has arity two.

The next kind in standard Haskell is `Type -> Type`, which is the kind of types that that have arity one. This is Haskell's version of a type constructor. It takes a type as input and produces a type. Theses are central in Haskell and will be explained further momentarily. For every arity of type constructor, there is an associated kind. For instance arity two type constructors have kind `Type -> Type -> Type`. It's natural to think of type constructors are type-level functions, and we will do so.

As should be visible already, types in Haskell are a complicated business. However it turns out that the structure of the type system in Haskell is well represented mathematically. That encoding is as a category.

6.1.1 Types as a Category

Haskell's development was motivated by an interest in lazy functional languages. A highly desirable trait for a lazy language to have is **referential transparency**, the property that a function application / call can be replaced with its return value without changing the computation. In a lazy language without referential transparency, it becomes extremely difficult to write code with side effects, as the side effect must either be expressed concretely via some sort of stream object, or modeled via continuation passing style programming. If a lazy language allows effectful statements without some method of tracking them, then programming consistent behavior becomes extremely difficult, as the effect will occur when the associated code is accessed rather than when it was defined. Recall that this is a direct result of the definition of a lazy language. Thus two reads from a file or from the user could be interpreted in either order depending not on when they were inputted but when their content was inspected. This is obviously highly undesirable. The other methods of effect tracking were functional but unwieldy.

Several major versions after the initial release of Haskell, monads were introduced to the Haskell type system. At a high level, a effectful computation in Haskell is represented by a sequence of monadic types. The monad contains as the categorical object the standard return type of the computation, as it if were not effectful, and the effect (critically the order of the effects) is captured in monoidal element. An example appears later in this chapter for the state monad. This encoding of the nature and effectfulness of a computation in the type system leverages the already extremely powerful type system while providing a flexible and convenient solution for modeling computations that would otherwise rely on information outside of the type system.

Modeling effectful computation is critical to any programming language, as without it, a program can't interact with anything outside itself. It can't take in any information on which to process, and it can't output any information that it may have computed. Naturally, if one writes a program that computes some value, one is likely interested in said value. Without effects, that value is inaccessible.

Embedding the tracking of effects in the type system is both natural and powerful because the type system is what maintains the safety and correctness of the lazy evaluation that is central to Haskell.

Monads in Haskell are slightly different than those from standard category theory.

This section will delineate some of those differences and lay out the basic functionality of monads in Haskell. In general, Haskell's monads allow one to compose functions within some context. What that context is depends on the monad.

Consider the category *Hask*. The objects of *Hask* are the full collection of Haskell types (we can think of types here as possibly infinite sets of possible values). The morphisms are functions (excluding type constructors), and function composition is associative with `.` as Haskell's function composition operator. This description of *Hask* as a category is light and not rigorous, as the full rigor version requires limiting to terminating programs, restrictions on kinds and other details that are out of scope for this thesis.

Functors and monads as introduced in the Prerequisites chapter 1.5 are relevant to Haskell. Particularly of interest are monads.

A functor is a mapping between two categories that preserves the identity morphism for each object and respects the composition of morphisms. Thus for a functor $F{:}C \to D$, $F(id_c) = id_{F(c) \in D}$. We are only interested in endofunctors on *Hask*, functors that go from *Hask* to itself. Thinking about what the means, we find that we are looking for something that takes a type (an object in *Hask*), and produces another type, subject to some rules. That should sound familiar, as type constructors are exactly that: something that takes a type and produces a type. So looking for our endofunctors in *Hask*, we are now looking for a type constructor (something with kind `Type -> Type`). However we also need our type constructor to respect composition and identity.

As a reminder, we want $F(g \circ h) = F(g) \circ F(h)$ for our functor F and g, h morphisms (Haskell functions). Consider some arbitrary type `a`. Our functor `f` takes every type to another type, so `f a` must also be some type. We won't give it a name, and simply call it `f a`. We are looking for some function `fmap` that takes morphisms `a -> b` and produces morphisms `f a -> f b` that satisfy `fmap id = id` (the identity) and `(fmap f) . (fmap g) = fmap (f . g)`.

That's exactly what Haskell does. A type constructor `f` implements `Functor` if it provides some a function `fmap :: (a -> b) -> f a -> f b` for arbitrary `a`, `b`. Unfortunately the identity and composition properties must be checked and assured by the programmer.

Let's motivate this categorical Haskell with an example. There is a type constructor `[]` which for any type constructs a list of that type. A list in Haskell is an ordered sequence of values all sharing the same type. So `[a]` is a list of values of type `a`. Consider that you have a list of items, and you what to apply some function to each element, resulting in a new list in which each element in order is the corresponding element in the input list with the function applied to it. This "mapping" over a list is a common recurring pattern of computation. With some thought, we can see that `map`, which does exactly what we just described is really `fmap` specialized to a list. Given a function on each element, it produces a function on lists with the same behavior. This "lifting" of functions to work on structures is a central concept of idiomatic Haskell.

The other central categorical idea present in Haskell is the structure of a monad

Various. A monad is a monoid in the category of endofunctors of a fixed category, here *Hask*. So the appropriate thing to look for is some type constructor that has an associative operation and an unit that behaves appropriately with the associative operation. To start, every monad in Haskell is a functor. So our monad `m` must have some `fmap :: (a -> b) -> m a -> m b` as described above. Being a functor means that functions between internal types can be lifted to the wrapped monad type. It also means that composing functions and then lifting is equivalent to lifting and then composing.

Consider the class of objects of form `a -> m b` in the category of endofunctors on *Hask*. Putting aside the implementation details of how one makes a `m b` value from a `a` value (which is likely unique to each monad), it is useful to give names to two functors of interest. The first is `return :: a -> m a`. This is perhaps the most natural function that produces a monadic value. Knowing how to make some `m a` value from a `a` value is a must for any useful monad. What Haskell calls `return` is what was called η in the monad definition given in the prerequisites chapter 1.5. The other is `bind :: m a -> (a -> m b) -> m b`, (previously μ) also commonly written as an infix operator `>>=`. Thinking of the monad as "wrapping" the internal type `a` is a useful mindset here. From that point of view, `bind` says if one has a wrapped value, and one can make another wrapped value of possibly different type given a value of the internal type `a`, then it's natural to be able to apply the function to the wrapped value, "unwrapping" `m a` in the way that is suitable for the monad in the process.

Now we have our two components of a monoid, though we still have yet to show that they behave as we want. We want `bind` to act as our associative operation (think composition) between morphisms Various. Put simply, we want functions that look like `f :: a -> m b` and `g :: b -> m c` to be composable, and that composition should be associative. So just as `g' (f' x)` $\equiv$ `(g' . f') x` for standard functions `f' :: a -> b` and `g' :: b -> c` and `x :: a`, it's natural to want `(v >>= f) >>= g` $\equiv$ `v >>= (x -> (f x) >>= g)`. Note that `x -> (f x) >>= g` is Haskell's syntax for a lambda (anonymous function). The names on the left of the arrow (here `x`) are the arguments, and the expression on the right of the arrow is the body of the function. This associativity law basically says that it is equivalent to perform two actions in sequence or to merge to actions into one action and then apply that action.

Finally we have that `return` acts as a identity element for `>>=`. We have that `(return x) >>= f` $\equiv$ `f x` and that `v >>= return` $\equiv$ `v`. So `return` doesn't do anything once a value is already wrapped by the monad. Applying it as an action again doesn't nest wrappings, `return` simply takes a value and lifts it to be a monadic action.

As with Haskell's functors, these two laws must be manually assured by the programmer. However with them in place, monads act as we expect them to, and are well modeled categorically.

6.2 Pure Functional Programming

A **functional** programming language is one in which the primary method of computation is the calling of functions. Haskell is on the far end of the spectrum of languages from imperative to functional in that function calling is the only method of computation in Haskell. A **pure** functional language is one in which all the functions must be pure, namely have output dependent solely on the immediate inputs and neither observe nor modify any external state. A pure function can be understood as a function that will produce the same output for the same set of inputs regardless of context, and the execution of which will not cause any contextual value to change. Haskell unequivocally requires that all functions (and thus all computation) be pure. Again this pureness provides referential transparency, which is an extremely desirable trait for a lazy language to have.

Having functions be pure is desirable because, as often trumpeted by Haskell supporters, it ensures that computation is side effect free. This is chiefly in contrast with unintended or unforeseen side effects. A common cause of bugs in large libraries in some languages that do not enforce these restrictions is using functions that mutate some external state that is not accounted for by the programmer. If this causes problems down the line, it can be hard to track down the source of the error, as the culprit function call may not be obviously related or nearby in the code.

While avoiding side effects is safer, completely eliminating them also prevents their productive use. Besides being inconvenient for tasks where external state may be intuitive or efficient, it also eliminates some central features of programming, such as receiving input to the program or producing output. These operations are fundamentally impure, as the result of the program depends on the order that they were invoked. Naturally, eschewing input and output entirely is not viable. The solution comes to us in the form of monads.

6.2.1 Why Monads Matter

Arguably the biggest legacy of Haskell so far in the programming language space, monads are Haskell's solution to the problem of state in pure functional programs. **State** refers to contextual data that effects the computation, either directly as an input to some calculation or in selecting the computation to perform. We say that a computation is **stateful** or **stateless** if it has or lacks this property, respectively.

Haskell calls individual monad computations actions, and they represent delayed computations. Consider a computation that produces some value `v :: a` and uses some state of type `s`. We naturally want to interleave actions that consult or modify the state with pure computations that don't touch the state directly. Haskell represents this in monadic form by wrapping the immediate type of each computation in a monad that represents the context of a computation occurring in the presence of state, even if that computation itself isn't directly interacting with the state.

The Haskell type `State s a` is a monadic action of type `a` that is informed by some state of type `s`. The two basic operations of any monad exist, namely `return` and `>>=` (also called "bind"). The former has type `return :: a -> State s a`.

The later has type `(>>=) :: (State s a) -> (a -> State s b) -> (State s b)`. These allow you to lift pure computations to monadic actions and compose monadic actions, respectively. Naturally we also need ways to interact with the state. For our fixed state type `s`, we can use `get :: State s s` to produce the state as a value we can then compute with, and `put :: s -> State s ()` to replace the stored state with the argument.

These tools allow one to create a sequence of bind actions that together represent some computations that consults and modifies state, eventually producing some value and some final state. We can put this sequence to effect by calling `runState :: State s a -> s -> (a,s)`, which supplies the sequence of actions with some initial state and then produces the final outcome.

Haskell's State monad represents stateful actions as a function from a state to a pair of a state and a value. The monad is simply a nice way of wrapping that abstraction up in a way with familiar rules. When the computation is run, the distinction of external state and argument is erased, as internally the state is just another argument. The monad saves one from writing in a state argument every time, as the majority of the computations don't need to touch it directly.

Technical details aside, thinking of external state and the input and output of a program as a monadic context in which computations occur allows for a very practical way to handle the difficulties of those inherently sequential problems in a way that is consistent with the lazy nature of Haskell.

Monads in Haskell appear constantly and are central to writing powerful and idiomatic Haskell. For instance, once annotated with the information we reconstruct, each function of the binary being analyzed is fed into Crucible through the generator interface, which is a complex monad (really a stack of monads that act together as a large monad). Among its other utilities, this monad models the inherently stateful actions of reading and writing mutable registers. When inside this generator, one can write things like `readRegister r` and get the contents of the register that is suitable for the position in the program. The monad handles all the mutability of the program state and allows one to write (or in this case translate) familiar looking imperative assembly rather than juggling SSA forms and explicit side effects.

6.3 Dependently Typed Haskell

Now that we have a grasp of both standard Haskell and some type theory presented in terms of the lambda calculus, we can appreciate the true power and complexity of Haskell's type system. As mentioned at the end of the type theory chapter, Haskell is equivalent to the system $\lambda\omega$. It features type constructors and polymorphism, but at first glance not dependent types. One can't put a term in a type definition, that much is clear. However Haskell's polymorphism is more refined than that of the lambda cube formulation. In the lambda cube, each type in a lambda abstraction or type constructor is either a single concrete type, or a universal qualification over all possible types. In contrast, Haskell features type classes, which group types by some shared behavior, similar to traits in Rust. These type classes allow one to write

functions and type constructors that are qualified over all types in one or more classes, rather than all possible types. This distinction allows us to be far more precise with our type constraints.

A type is said to implement a type class if implements the minimal complete definition of that class. For instance, the class `Ord` captures the property of some type `a` being totally ordered. In order for `a` to be in the type class `Ord`, it must first be in the class `Eq`, which states that that values of type `a` can be compared for equality. Then the implementer must supply some definition of `compare` for values of type `a`. This function takes two `a` values and produces a verdict from equality, less than or equal to, or greater than or equal to. The implementer is responsible for maintaining the invariants and properties that the class expects: `compare` is actually a total ordering, `==` is an equivalence relation, etc.

This intermediate level of specificity allows for the emulation of dependent types in Haskell. Specifically the formulation in use by `parameterized-utils` Inc. [2024], the library providing the dependent definitions in Crucible, is via the concept of **singleton types**.

6.3.1 Singleton Types

A singleton type is a type with a single valid value. This explanation assumes that all values are not the bottom value, as it complicates the semantics, and complete and well formed programs do not produce bottom values. Haskell's unit type, `()` is a singleton type.

Singleton types are useful because they form a bridge between types and values. For a standard type with multiple values, knowing the type of some expression is not enough to know the value and thus how to operate on it, or how to interpret it. A conditional expression on some boolean for example. Knowledge that the condition is some boolean is not enough to decide which arm of the conditional is the correct one. A singleton type erases this distinction. Knowing the type of some expression of a singleton type is equivalent to knowing the value of the expression, as there is only one choice. This direction of implication, types acting like values, is the useful feature of singleton types in this context.

Consider the most fundamental and more relevant set of singleton types, the natural numbers. Haskell contains an infinite sequence of type level natural numbers, equipped with a polymorphic function that produces the corresponding value level natural, `natValue :: NatRepr n -> Natural`. If the inverse exists, then dependent types are emulatable, as to produce some type `B` indexed by a natural number value, one could simply produce the equivalent type level number and use a type constructor to parameterize `B` with the number at the type level.

While this idea is what happens in spirit, there is a core problem. It is impossible to correctly type the function from values to types in Haskell. Haskell requires each expression to have a known type. Let `toType` be our value to type converter for naturals. It's clear `toType 1` has a type, namely the type level 1. The same is true for every other value level natural input. However Haskell functions must produce a

single type, not a different type for each input. So this is not quite enough.

The answer lies in **existential types**. Returning to the types as proofs frame of mind, an existential typed value is a witness (read proof) of the existence of some type. Critically this internal type must exist in order to form an existential, but the resulting value does not contain the information of what the type was. At first glance, this seems useless. Why would one want to assert the existence of a type, and particularly if the type isn't then known? However that lack of information is exactly what makes existential types so useful.

Let us turn to the definition of existentials from the same library as before.

```
1 data Some (f :: k -> Type)
2   forall x. Some (f x)
```

The exact details are not important, but we see that `f` is some type constructor, and that to produce a `Some` value, `f x` must exist and be well typed. Thus a `Some` value witnesses `x` as a suitable input to `f`. However we see that while `Some` is parameterized (via type constructor) by `f`, it does not have a type level parameter for `x`. Thus the type of `x` is hidden. Critically, `Some f` is the resulting type for all suitable choices of parameter type `x`. This is a sort of type level funnel, it takes any suitable type `x` and produces a uniform type `Some f`.

Let us return to our example with naturals. Consider the definition of the type constructor for representations of naturals.

```
1 data NatRepr (n :: Nat)
```

This representation is a transparent cover of the type level natural `n`. However the key insight is that `NatRepr :: Nat -> Type`, exactly the shape of the input to `Some`. The same library provides `mkNatRepr :: Natural -> Some NatRepr`, which takes a value level natural and produces an existential type level natural.

With this we are halfway to what we want. We can define our ideally dependently typed function in terms of type level naturals, and we can produce a witness for the appropriate type level natural given a value level natural. However this isn't enough, as our type constructor takes a type level natural, not a witness of the existence of one. There is still one part missing.

It turns out that while `Some` erases the internal type, it doesn't erase the internal value. So matching on the existential produces the original `f x` inside. One must be careful however, since the type rules of Haskell require that all expressions be well typed at compile time. Thus the internal type `x`, the one we actually care about, cannot escape the context of the match. The type of the match expression must be some concrete type that is known at compile time. In general, one can use any function that is universally qualified in its input and concrete in its output, where the universe of inputs must be a superset of the suitable inputs to `f`. This pattern is called **projecting** out of the existential, as it produces a value that is not wrapped by an existential, and requires a internal function that is defined for all possible contents of the existential. This is again a sort of type level funnel, which takes all possible internal types `x` and produces a single known type. This is not terribly restrictive, though sometimes inconvenient. The library also provides utilities like comparing for

equality between existential types and type classes representing natural constraints.

A function on the naturals can be restricted to operate on some sub-interval by bounds created with a type-level less than or equal to operator. While taking type level arguments via `NatRepr` values, this encodes the exact same restriction as the subtraction example from the type theory section 5.2.4. In the same manner to apply the function to a natural, one first needs to provide a witness that the value is in that interval. In the lambda calculus, this is done by some lambda abstraction that transmutes it to the appropriate type encoding that information. In dependent Haskell via singletons, this is achieved with natural number constraints. For example selecting some bits from a bit vector Inc. [2023].

```
1 BVSelect :: (1 <= w, 1 <= len, idx + len <= w)
2           => !(NatRepr idx)
3           -> !(NatRepr len)
4           -> !(NatRepr w)
5           -> !(f (BVType w))
6           -> App ext f (BVType len)
```

Again the exact details aren't important here, but notice the first line that contains restrictions that the various arguments are reasonable. The input and output vectors are non-zero in length, the length of selection and the offset together shouldn't go past the end of the input. These are sanity checks that traditionally happen at the programmer's discretion. By encoding them as type level constraints, it is impossible to produce an undefined output via any combination of inputs, as unsuitable combinations are not well typed.

Conclusions

> When we asked Pooh what the opposite of an Introduction was, he said "The what of a what?" which didn't help us as much as we had hoped, but luckily Owl kept his head and told us that the Opposite of an Introduction, my dear Pooh, was a Contradiction; and, as he is very good at long words, I am sure that that's what it is.
>
> A. A. Milne

This chapter unsurprisingly is a concise statement of the output of this thesis along with some related information such as next steps.

The primary output of the thesis aside from this document is the pair of code artifacts. These are the P-Code dumping plug-in for Ghdira (Ulmer [2023b]) and the main Haskell code that forms the backend for Crucible (Ulmer [2023a]). The majority of the work of the thesis was the formation of the Haskell code, along with learning the many related pieces of theory.

There are a number of natural next steps for continued work on this topic. These primarily consist of extending the backend to accommodate more sophisticated programs, along with expressing the full generality of P-Code. Examples of extensions include tracing memory accesses with a write-log to allow for simple memory modeling and generalizing the VarNode handling code to represent operations which involve multiple contiguous registers. In addition, Crucible for P-Code currently lacks an interface for users to meaningfully interact with the symbolic representation, as the standard usage of Crucible is via tools like Crux that work via source code embeddings. This approach is unsuitable for use on binaries that lack accompanying source code, and thus doesn't apply to the primary use-case of this backend. Alternative methods could include an interface in one of the other tools built on Crucible such as Galois's Software Analysis Workbench.

Appendices

Appendix A

RISC-V Reference

This appendix is a largely complete detailing of the RISC-V ISA. It contains descriptions of the core components of the ISA followed by some annotated examples.

A.1 Concrete Example Architecture

For the purpose of gaining an understanding of the inner workings of a computer at the level that will be discussed in this work, this section provides a summary of a concrete architecture. The architecture in question is the RISC-V architecture, as it is simple and modern as well as being an open standard. This section can be safely skipped by anyone who has worked with assembly before or has a general sense of how assembly languages work. Useful alternatives to this section for motivating and detailing assembly languages include Knuth [2005] and Abelson and Sussman [1996] section 5.1. Both describe alternative architectures, but do so from different standpoints. The first of the two specifically is noted, as it also includes an explanation of the binary encoding of integers for those readers who want it.

What follows is a summary the RISC-V instruction set architecture for the base standard. Thus it is the simplest configuration of RISC-V. The details of the modular design of the RISC-V standard are out of the scope of this paper. The rest of this section summarizes the relevant RISC-V standard.Foundation [2019]

The core of the RISC-V specification is the Integer Instruction Set Architecture. This is the only part of the specification that is of interest to us, so it will be referred to as the ISA or the RISC-V ISA from now on. In addition, we will be concerned only with the 32 bit version. There is a 64 bit version, but it is also not of interest for our purposes.

The ISA is composed of descriptions of a number of instructions, each of which performs some small computation or data movement. They can be performed in sequence to perform more complicated computations. The ISA also contains a number of auxiliary details that will be included in this summary only when relevant.

A.1.1 Layout

In order to understand what each instruction does, some knowledge of the parts of the computer is necessary.

A computer conforming to the RISC-V ISA has a set of locations for storing integers. These locations are separated into two groups. The first group are the registers. There are 32 registers, each of which is 32 bits wide. These registers are numbered from `x0` to `x31`, and also have more user friendly names. The register `x0` always has the value zero for convenience. There is also a separate register, `pc`, or *program counter*, which cannot be used for general computation and has special meaning.

The second group is called memory. It is composed of a large number of locations for storing 8 bit values. These locations are indexed by 32 bit numbers, and the index of a location is called an address. One often makes reference to several locations with neighboring indices, treating them as one larger location for storing 16 or 32 bits of data instead. The exact details of the equivalence of between these two representations is out of scope for this background, but for a given address, the 8 bits stored in memory for that address are the same as the least significant 8 bits of the 16 bit data location referenced with that address. The same relationship exists between all pairs of 8, 16, and 32 bit data widths for a given address.

A.1.2 Instruction Background

In general, a single instruction in this ISA references from one to three data registers and zero to one fixed integer value (called an Immediate) according to the instruction. The computer performs some action according to the exact instruction and the arguments. The exact details of that encoding are not important for this paper and thus will not be covered.

To execute a single instruction a RISC-V computer does the following:

- Read a 32 bit value from memory at the address stored in the program counter `pc`.
- Translate that value into an instruction and choices of registers for its arguments according to the above mentioned encoding.
- Do one of the following, with a register as the destination unless otherwise specified:
  - Acquire another value from memory according to the instruction and arguments.
  - Perform the calculation according to the instruction and arguments.
  - Write a value to memory according to the instruction and arguments.
  - Write a value to a register according to the instruction and arguments.
- Increment `pc` by a single instruction (4, as memory is indexed in 8 bit units).

By repeating the above cycle, a sequence of instructions encoded in 32 bit values in memory will be executed in order, as `pc` will be incremented over them, one per cycle. Thus the effects of instructions can be composed and the state of the general purpose registers and the memory can change in complicated and semantically meaningful ways over the course of several instructions.

One detail of the instruction encoding is the presence of *immediate values*. An immediate value is a constant number that is encoded directly in the instruction. For instance, adding 4 to a register, and storing the result in a second register is one instruction, and the literal 4 is an immediate.

There are several formats in which instructions are encoded in binary depending on the nature of the arguments, but in general there is a span of bits that indicates the layout, a span that indicates the operation, and several spans for indicating the inputs and outputs. For instance an instruction that operates on registers might have three 5 bit fields that each indicate one of the 32 registers, two for inputs and one for the output. An instruction that takes an immediate might have a span (often at the end of the instruction) of 12 bits for encoding the desired constant value. The ordering and breakdown of the fields is not terribly relevant and has been elided here, though it is comprehensively documented in the specification.

What follows is a list of the instructions in the base ISA. These definitions are included to give a sense of how much one instruction does in the RISC-V ISA. The reader is expected to get a general sense, but not remember the exact semantics of each instruction. Nuances of particular instructions will be explained in greater depth as needed later.

A.1.3 Integer Computational Instructions

A computational instruction in the RISC-V ISA is one that performs some simple operation on registers or immediates and places the result in a register.

Operation	Description
`ADDI`	Takes a source register, a destination register, and a signed 12 bit immediate. Sets the value of the destination register to the source register plus the immediate. This instruction is useful for copying registers to other registers by adding zero, or loading an immediate into a register by adding the zero register and the immediate.
`SLTI`	(Set Less than Immediate) Takes a source register, a destination register, and a signed 12 bit immediate. Sets the value of the designation register to 1 if the source register is less than the immediate when both are treated as signed numbers.
`SLTU`	(Set Less than Unsigned Immediate) Operates the same way as `SLTI` but does the comparison treating both as unsigned numbers. Note that the immediate is still sign extended, so passing a negative immediate may result in unintended behavior.

`ANDI`
`ORI`
`XORI`
	Logical bitwise operations that take a source register, a destination register, and a 12 bit signed immediate and write the result of the respective binary operation applied to each bit of the source register and the sign extended immediate to the destination register. Note that `XORI` with -1 is equivalent to a bitwise logical `NOT`.

`SLLI`
`SRLI`
`SRAI`
	Bit shifts left and right that take a source and destination register and a 5 bit unsigned immediate. The source register shifted appropriately by the amount provided in the immediate. The two versions of right shifts are Logical and Arithmetic, which differ in that Arithmetic copies the original sign bit from the source register to the destination register.

`LUI`	Takes a destination register and a 20 bit immediate. The immediate is placed in the upper 20 bits of the register, and the bottom 12 bits are filled with zeros. This instruction combined with `ADDI` is intended to be used to build 32 bit constants.
`AUIPC`	Takes a destination register and a 20 bit immediate. Forms a 32 bit constant by taking the immediate as the upper 20 bits and filling the lower 12 bits with zero, then adds that 32 bit number to `pc` as an unsigned addition and writes that sum to the destination register. Note that `pc` here is the same value as the address of this instruction in memory. This instruction, again in combination with `ADDI` is used to make `pc` relative addresses, the usefulness of which is not in scope for this definition.

`ADD`
`SUB`
`SLT`
`SLTU`
`AND`
`OR`
`XOR`
`SLL`
`SRL`
`SRA`
	Take two source register and a destination register and act in a similar manner as their immediate counterparts, simply taking the second 32 bit source register instead of the 12 bit immediate.

`NOP`	An instruction that does nothing. This is encoded as `ADDI x0, x0, 0`, understood as adding zero to zero and discarding the results.

A.1.4 Control Transfer Instructions

A control transfer instruction alters the sequence of instructions being executed. Specifically, it changes `pc` such that the next instruction to be executed is not the immediate successor to the current instruction. Control flow instructions are used for calling subroutines, loops, and conditional blocks.

Operation	Description
`JAL`	An unconditional jump. Takes an immediate and a destination register. The 20 bit immediate is sign extended to 32 bits and multiplied by two. The resulting value is then added to `pc`. The address of the next sequential instruction (`pc`+4) is stored in the destination register. Thus in combination to the incrementing of `pc` that happens on every instruction, has the effect of making the next instruction that is executed one at the given offset from what it would normally be. In addition, storing the original next instruction's location in a register allows for returning to the original control path. Used for making subroutine calls. `CALL` is shorthand for `JAL` with the standard choice of register.
`JALR`	Another unconditional jump. Takes a source and destination register and a 12 bit signed immediate. The immediate is added to the source register and `pc` is written with the result. Note that this version is not relative to `pc`. Once again, the original next instruction location is written to the destination register. Used for calls with targets not known at compile time and returning from subroutines by using the caller location saved in `JAL` and discarding the jump location by setting the destination register to `x0`. `RET` is short for return to the standard return address register, and discard the current location by using `x0` for the link register.

`BEQ`
`BNE`
`BLT`
`BLTU`
`BGE`
`BGEU`
	Conditional branches. Take a 12 bit signed immediate and two source registers. Perform some function from the two registers to a boolean, and if the value is true, then it performs a `pc`-relative jump with the immediate as the offset. If the function evaluates to false on the two registers, then the next instruction is executed as normal. The functions for each instruction are
	• `BEQ` checks that the two registers are equal.
	• `BNE` checks that the two registers are not equal.
	• `BLT` checks that the first register is less than the second.
	• `BLTU` does the same, but treating the registers as unsigned values.
	• `BGE` checks that the first register is greater or equal to the second.
	• `BGEU` does the same, but treating the registers as unsigned values.

A.1.5 Load and Store Instructions

A load or store instruction is an instruction that moves data between registers and memory.

Operation	Description
`LOAD`	Takes a source and destination register and a signed 12 bit immediate. Adds the source register and the immediate and treats the sum as an address in memory. Depending on the variant, 8, 16, or 32 bits in memory at that address are written to the destination register's least significant bits. The details of loading different sized values, and interpreting them as either signed or unsigned is not critical to our intuition, and has thus been excluded.
`STORE`	Takes two source register and a 12 bit signed immediate. Adds the first source register and the immediate, treats the sum as an address in memory, and writes the value of the second source register to that location in memory. Again, there are variants for different widths of data, 8, 16, and 32 bits specifically. In the case where not all of the register is copied, the least significant bits of the first register are copied to memory.

A.1.6 Memory Ordering, Environment Calls, Breakpoints, and Hints

All of the above are included in the base RISC-V specification, but are not relevant to our purposes and are thus not included here.

A.2 Concrete RISC-V Assembly Examples

In order to build familiarity with the concepts of assembly and instructions, a few examples are included here. A reader unfamiliar with assembly, or programming in general, is encouraged to read them carefully and ideally step through them in one's mind for a simple case or two. They are unoptimized, but are idiomatic and thus serve as good introductory examples. I will be using the standard RISC-V assembly syntax, where the destination register comes first, and the sources come after it, though this decision is arbitrary. The remainder of any line after a hash (#) is a comment and will be ignored. They are simply included for the purpose of explanation.

First consider a function that takes a non-negative whole number n and returns the n-th Fibonacci number. A natural if not fast way to calculate that is to simply follow the rule of each number being the sum of the previous two until we reach the index we were asked for:

Listing A.1: Fibonacci

```
1  # Let's assume we are given the index in register a0, and the zeroth
2  # and first Fibonacci numbers are 0 and 1 respectively
3  fibonacci:
4  mv t0, x0
5  mv t1, 1          # Set up initial values
6  loop:
7  beqz a0, done     # If we are at the indicated index, we can stop
8                    # iterating
9  add t2, t1, t0    # t2 is our next value
10 mv t0, t1
11 mv t1, t2         # Shift over our saved elements so we can do it again
12                   # if we need to
13 addi a0, a0, -1   # Decrement a0, we have taken one step
14 j loop            # Take us back to the loop label, don't save
15                   # infomation to return here
16 done:
17 mv a0, t0         # The caller expects the answer in the a0 register
18 ret               # Return to caller
```

Note the core idea is here is do the most basic step in a generic manner, here lines 9 through 11, and surround them with structure such that they can be repeated (lines 7, 13, and 14).

Now consider a slightly more complicated example: primality testing. Our method here will be to test if pairs of numbers multiply to our given number n, and minimize the number of pairs we have to test. We know that multiplication is commutative and that for positive inputs, it is monotonically increasing in both operands. That means that we only need to test one of each $a \cdot b$ and $b \cdot a$ pair, and that if ever $a \cdot b$ is greater than n, we can stop this line of tests and start another. So we will try pairs $a \cdot b$, and increase b until either we find a match or it becomes too big. If it is too big, we will increase a, and start testing again with b initially equal to a. We can stop incrementing a if it is equal to n. At that point, we know that the input is prime. Note that this is an extremely inefficient algorithm and implementation.

Listing A.2: Primality Test

```
1  # Our number to test is in register a0, and the caller expects the
2  # result, 1 if it is prime, 0 otherwise, in a0. We will want to call
3  # other things, so we need to save a few numbers out of the way. We
4  # make space for them on the stack, a place in memory for storing
5  # temporary values.
6  isPrime:
7  addi sp, sp, -16     # Make space for 4 4 byte values
8  li t0, 2             # Initially a = 2
9  outer_loop:
10 mv t1, t0
11
12 inner_loop:
13 st ra, (sp)
14 st t0, 4(sp)
15 st t1, 8(sp)
16 st a0, 12(sp)        # Registers that might get clobbered are safely
17                      # saved
18
19 mv a0, t0
20 mv a1, t1
21 call multiply
22 # This is another function we are calling out to. It takes two numbers
23 # in a0 and a1 and returns their product in a0. It's definition has
24 # been elided for brevity. In most real computers, there would be an
25 # instruction for this, but the most minimal RISC-V set does not have
26 # one.
27
28 mv t2, a0            # This is our product to test
29 ld ra, (sp)
30 ld t0, 4(sp)
31 ld t1, 8(sp)
32 ld a0, 12(sp)        # Restore our saved values
33
34 beq a0, t2, found   # Our product matches
35 bgt t3, a0, too_big # Our product is too large
36 addi t1, t1, 1      # Need to keep going, increment b and try again
37 j inner_loop
38
39 found:              # We found a counter example
40 mv a1, x0           # Not prime
41 addi sp, sp, 12     # De-allocate the space we allocated
42 ret
43
44 too_big:           # Our product was too big, are there more to test?
45 beq t0, a0, done    # a = n, we are done
46 addi t0, t0, 1
47 j outer_loop
48
49 done:              # Failed to find any counterexamples
50 mv 1, a0           # Is prime
51 addi sp, 12, sp     # De-allocate the space we allocated
52 ret
```

Works Cited

H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996. ISBN 0262011530.

N. S. Agency. *P-Code Operation Reference*, 2019.

H. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991. doi: 10.1017/S0956796800020025.

L. Barrett, 2021. URL `https://galois.com/blog/2021/10/under-constrained-symbolic-execution-with-crucible/`.

R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, oct 1991. ISSN 0164-0925. doi: 10.1145/115372.115320. URL `https://doi.org/10.1145/115372.115320`.

N. S. A. R. Directorate, 2023. URL `https://ghidra-sre.org`.

R. Dockins, A. Foltzer, J. Hendrix, B. Huffman, D. McNamee, and A. Tomb. Constructing semantic models of programs with the software analysis workbench. volume 9971, pages 56–72, 07 2016. ISBN 978-3-319-48868-4. doi: 10.1007/978-3-319-48869-1_5.

R.-V. Foundation. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*, 2019.

HackOvert. Ghidrasnippets. `https://github.com/HackOvert/GhidraSnippets#dumping-raw-pcode`, 2023.

S. B. Horwitz. Lecture Notes. URL `https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/1.LAMBDA-CALCULUS.html`.

W. Howard. The formulae-as-types notion of construction. 1980.

G. Inc. Crucible, 2023. URL `https://github.com/GaloisInc/crucible/blob/0d26eb423fa613ccf0410a84f5c72a7d7473b6be/crucible/src/Lang/Crucible/CFG/Expr.hs/#L641`.

G. Inc. parameterized-utils, 2024. URL `https://hackage.haskell.org/package/parameterized-utils`.

D. E. Knuth. *The art of computer programming. Volume 1, fascicle 1, MMIX A RISC computer for the new millennium.* Addison-Wesley, Upper Saddle River, NJ, 1st edition edition, 2005. ISBN 0-321-63736-4.

S. Messick. Limits in category theory. 01 2007.

P. S. Mulry. Monads in semantics. *Electronic Notes in Theoretical Computer Science*, 14:275–286, 1998. ISSN 1571-0661. doi: https://doi.org/10.1016/S1571-0661(05)80241-5. URL `https://www.sciencedirect.com/science/article/pii/S1571066105802415`. US-Brazil Joint Workshops on the Formal Foundations of Software Systems.

N. Naus, F. Verbeek, D. Walker, and B. Ravindran. A formal semantics for p-code. In A. Lal and S. Tonetta, editors, *Verified Software. Theories, Tools and Experiments.*, pages 111–128, Cham, 2023. Springer International Publishing. ISBN 978-3-031-25803-9.

niconaus. Pcode-dump. `https://github.com/niconaus/PCode-Dump`, 2022.

M. I. Schwartzback. Lecture notes on static analysis. `http://www.itu.dk/people/brabrand/UFPE/Data-Flow-Analysis/static.pdf`.

T. Ulmer. Crucible-pcode. `https://github.com/TCCQ/crucible-pcode`, 2023a.

T. Ulmer. Pcode-dump. `https://github.com/tccq/PCode-Dump`, 2023b.

Various. Haskell monad laws. URL `https://wiki.haskell.org/Monad_laws`.

K. N. W. Kahan. `https://www.netlib.org/fdlibm/e_sqrt.c`, 1986.